MSP50C6xx Mixed-Signal Processor User's Guide

Mixed Signal Products

SPSU014A

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Preface

Read This First

About This Manual

This user's guide gives information for the MSP50C6xx mixed-signal processor. This information includes a functional overview, a detailed architectural description, device peripheral functional description, assembly language instruction listing, code development tools, applications, customer information, and electrical characteristics (in data sheet).

How to Use This Manual

This document contains the following chapters:

- Chapter 1 –Introduction to the MSP50C6xx
- Chapter 2 –MSP50C6xx Architecture
- Chapter 3 –Peripheral Functions
- Chapter 4 –Assembly Language Instructions
- Chapter 5 –Code Development Tools
- Chapter 6 –Applications
- Chapter 7 –Customer Information
- Appendix A –Additional Information

Notational Conventions

This document uses the following conventions.

- Program listings, program examples, and interactive displays are shown in a `special typeface` similar to a typewriter's. Examples use a **`bold version`** of the special typeface for emphasis; interactive displays use a **`bold version`** of the special typeface to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.).

Here is a sample program listing:

```
0011  0005  0001         .field    1, 2
0012  0005  0003         .field    3, 4
0013  0005  0006         .field    6, 3
0014  0006               .even
```

Here is an example of a system prompt and a command that you might enter:

```
C:  csr –a /user/ti/simuboard/utilities
```

- In syntax descriptions, the instruction, command, or directive is in a **bold typeface** font and parameters are in an *italic typeface*. Portions of a syntax that are in **bold** should be entered as shown; portions of a syntax that are in *italics* describe the type of information that should be entered. Here is an example of a directive syntax:

  .asect **"***section name***",** *address*

  .asect is the directive. This directive has two parameters, indicated by *section name* and *address*. When you use .asect, the first parameter must be an actual section name, enclosed in double quotes; the second parameter must be an address.

- Square brackets (**[** and **]**) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets; you don't enter the brackets themselves. Here's an example of an instruction that has an optional parameter:

  LALK *16–bit constant [, shift]*

  The LALK instruction has two parameters. The first parameter, *16-bit constant*, is required. The second parameter, *shift*, is optional. As this syntax shows, if you use the optional second parameter, you must precede it with a comma.

  Square brackets are also used as part of the pathname specification for VMS pathnames; in this case, the brackets are actually part of the pathname (they are not optional).

- Braces ({ and }) indicate a list. The symbol **|** (read as *or*) separates items within the list. Here's an example of a list:

  ```
  { * | *+ | *– }
  ```

  This provides three choices: `*`, `*+`, or `*–`.

  Unless the list is enclosed in square brackets, you must choose one item from the list.

- Some directives can have a varying number of parameters. For example, the .byte directive can have up to 100 parameters. The syntax for this directive is:

.byte *value$_1$ [, ... , value$_n$]*

This syntax shows that .byte must have at least one value parameter, but you have the option of supplying additional value parameters, separated by commas.

Information About Cautions and Warnings

This book may contain cautions and warnings.

This is an example of a caution statement.

A caution statement describes a situation that could potentially damage your software or equipment.

This is an example of a warning statement.

A warning statement describes a situation that could potentially cause harm to you.

The information in a caution or a warning is provided for your protection. Please read each caution and warning carefully.

Trademarks

Intel, i486, and Pentium are trademarks of Intel Corporation.

Microsoft, Windows, Windows 95, and Windows 98 are registered trademarks of Microsoft Corporation.

Contents

1 Introduction to the MSP50C6xx 1-1
1.1 Features of the MSP50C6xx 1-2
1.2 Applications 1-3
1.3 Development Device: MSP50P614 1-4
1.4 Functional Description for the MSP50C614 1-5
1.5 MSP50C601, MSP50C604, and MSP50C605 1-9

2 MSP50C6xx Architecture 2-1
2.1 Architecture Overview 2-2
2.2 Computation Unit 2-5
2.2.1 Multiplier 2-5
2.2.2 Arithmetic Logic Unit 2-7
2.3 Data Memory Address Unit 2-11
2.3.1 RAM Configuration 2-12
2.3.2 Data Memory Addressing Modes 2-13
2.4 Program Counter Unit 2-14
2.5 Bit Logic Unit 2-14
2.6 Memory Organization: RAM and ROM 2-15
2.6.1 Memory Map 2-15
2.6.2 Peripheral Communications (Ports) 2-16
2.6.3 Interrupt Vectors 2-18
2.6.4 ROM Code Security 2-19
2.6.5 Macro Call Vectors 2-22
2.7 Interrupt Logic 2-22
2.8 Clock Control 2-26
2.8.1 Oscillator Options 2-26
2.8.2 PLL Performance 2-26
2.8.3 Clock Speed Control Register 2-28
2.8.4 RTO Oscillator Trim Adjustment 2-29
2.9 Timer Registers 2-31
2.10 Reduced Power Modes 2-33
2.11 Execution Timing 2-40

3 Peripheral Functions 3-1
3.1 I/O 3-2
3.1.1 General-Purpose I/O Ports 3-2
3.1.2 Dedicated Input Port F 3-4
3.1.3 Dedicated Output Port G 3-5
3.1.4 Branch on D Port 3-6
3.1.5 Internal and External Interrupts 3-7
3.2 Digital-to-Analog Converter (DAC) 3-9
3.2.1 Pulse-Density Modulation Rate 3-9
3.2.2 DAC Control and Data Registers 3-9
3.2.3 PDM Clock Divider 3-11
3.3 Comparator 3-15
3.4 Interrupt/General Control Register 3-18
3.5 Hardware Initialization States 3-20

4 Assembly Language Instructions 4-1
4.1 Introduction 4-2
4.2 System Registers 4-2
4.2.1 Multiplier Register (MR) 4-2
4.2.2 Shift Value Register (SV) 4-2
4.2.3 Data Pointer Register (DP) 4-2
4.2.4 Program Counter (PC) 4-2
4.2.5 Top of Stack, (TOS) 4-3
4.2.6 Product High Register (PH) 4-4
4.2.7 Product Low Register (PL) 4-4
4.2.8 Accumulators (AC0–AC31) 4-4
4.2.9 Accumulator Pointers (AP0–AP3) 4-5
4.2.10 Indirect Register (R0–R7) 4-5
4.2.11 String Register (STR) 4-6
4.2.12 Status Register (STAT) 4-6
4.3 Instruction Syntax and Addressing Modes 4-8
4.3.1 MSP50P614/MSP50C614 Instruction Syntax 4-8
4.3.2 Addressing Modes 4-9
4.3.3 Immediate Addressing 4-13
4.3.4 Direct Addressing 4-14
4.3.5 Indirect Addressing 4-15
4.3.6 Relative Addressing 4-16
4.3.7 Flag Addressing 4-19
4.3.8 Tag/Flag Bits 4-20

4.4 Instruction Classification 4-22
4.4.2 Class 2 Instructions: Accumulator and Constant Reference 4-28
4.4.3 Class 3 Instruction: Accumulator Reference 4-30
4.4.4 Class 4 Instructions: Address Register and Memory Reference 4-34
4.4.5 Class 5 Instructions: Memory Reference 4-36
4.4.6 Class 6 Instructions: Port and Memory Reference 4-38
4.4.7 Class 7 Instructions: Program Control 4-39
4.4.8 Class 8 Instructions: Logic and Bit 4-41
4.4.9 Class 9 Instructions: Miscellaneous 4-42
4.5 Bit, Byte, Word and String Addressing 4-44
4.6 MSP50P614/MSP50C614 Computational Modes 4-49
4.7 Hardware Loop Instructions 4-53
4.8 String Instructions 4-55
4.9 Lookup Instructions 4-57
4.10 Input/Output Instructions 4-59
4.11 Special Filter Instructions 4-59
4.12 Conditionals 4-69
4.13 Legend 4-70
4.14 Individual Instruction Descriptions 4-74
4.15 Instruction Set Encoding 4-189
4.16 Instruction Set Summary 4-198

5 Code Development Tools 5-1
5.1 Introduction 5-2
5.2 MSP50C6xx Development Tools Guidelines 5-4
5.2.1 Categories of MSP50Cxx Development Tools 5-4
5.2.2 Tools Definitions 5-5
5.2.3 Documentation 5-8
5.3 MSP50C6xx Code Development Tools 5-8
5.3.1 System Requirements 5-8
5.3.2 Hardware Tools Setup 5-9
5.4 Assembler 5-11
5.4.1 Assembler Directives 5-11
5.5 C– – Compiler 5-16
5.5.1 Foreword 5-16
5.5.2 Variable Types 5-17
5.5.3 External References 5-17
5.5.4 C– – Directives 5-18
5.5.5 Include Files 5-19
5.5.6 Function Prototypes and Declarations 5-21
5.5.7 Initializations 5-21
5.5.8 RAM Usage 5-21
5.5.9 String Functions 5-22
5.5.10 Constant Functions 5-23

5.6 Implementation Details 5-24
5.6.1 Comparisons 5-24
5.6.2 Division 5-26
5.6.3 Function Calls 5-26
5.6.4 Programming Example 5-27
5.6.5 Programming Example, C –– With Assembly Routines 5-29
5.7 C– – Efficiency 5-37
5.7.1 Real Time Clock Example 5-39
5.8 Beware of Stack Corruption 5-57
5.9 Reported Bugs With Code Development Tool 5-58

6 Applications 6-1
6.1 Application Circuits 6-2
6.2 Initializing the MSP50C6xx 6-4
6.2.1 File init.asm 6-5
6.3 TI-TALKS Example Code 6-8
6.4 RAM Overlay 6-9
6.4.1 RAM Usage 6-9
6.4.2 RAM Overlay 6-10
6.4.3 Adding Customer Variables 6-10
6.4.4 Common Problems 6-11

7 Customer Information 7-1
7.1 Mechanical Information 7-2
7.1.1 Die Bond-Out Coordinates 7-2
7.1.2 Package Information 7-2
7.2 Customer Information Fields in the ROM 7-11
7.3 Speech Development Cycle 7-12
7.4 Device Production Sequence 7-12
7.5 Ordering Information 7-14
7.6 New Product Release Forms (NPRF) 7-14

A Additional Information A-1
A.1 Additional Information A-2

Figures

1–1 Functional Block Diagram for the MSP50C614/MSP50P614 1-5
1–2 Oscillator and PLL Connection 1-7
1–3 RESET Circuit 1-8

2–1 MSP50C6xx Core Processor Block Diagram 2-3
2–2 Computational Unit Block Diagram 2-4
2–3 Overview of the Multiplier Unit Operation 2-7
2–4 Overview of the Arithmetic Logic Unit 2-9
2–5 Overview of the Accumulators 2-10
2–6 Data Memory Address Unit 2-12
2–7 C6xx Memory Map (not drawn to scale) 2-16
2–8 Interrupt Initialization Sequence 2-25
2–9 PLL Performance 2-27
2–10 Instruction Execution and Timing 2-40

3–1 PDM Clock Divider 3-11
3–2 Relationship Between Comparator/Interrupt Activity and the TIMER1 Control 3-16
4–1 Top of Stack (TOS) Register Operation 4-3
4–2 Relative Flag Addressing 4-19
4–3 Data Memory Organization and Addressing 4-45
4–4 Data Memory Example 4-47
4–5 FIR Filter Structure 4-59
4–6 Setup and Execution of MSP50P614/MSP50C614 Filter Instructions, N+1 Taps 4-67
4–7 Filter Instruction and Circular Buffering for N+1 Tap Filter 4-68
4–8 Valid Moves/Transfer in MSP50P614/MSP50C614 Instruction Set 4-132

5–1 10-Pin IDC Connector (top view looking at the board) 5-3
5–2 Hardware Tools Setup 5-10

6–1 Minimum Circuit Configuration for the C614/P614 Using a Resistor-Trimmed Oscillator 6-2
6–2 Minimum Circuit Configuration for the C614/P614 Using a Crystal-Referenced Oscillator 6-3

7–1 100-Pin QFP Mechanical Information 7-7
7–2 64-Pin QFP Mechanical Information 7-8
7–3 120-Pin, Grid Array Package for the Development Device, MSP50P614 7-9
7–4 Bottom View of 120-Pin PGA Package of the MSP50P614 7-10
7–5 Speech Development Cycle 7-12

Tables

2–1 Signed and Unsigned Integer Representation 2-5
2–2 Summary of MSP50C614's Peripheral Communications Ports 2-17
2–3 Programmable Bits Needed to Control Reduced Power Modes 2-36
2–4 Status of Circuitry When in Reduced Power Modes (Refer to Table 2–3) 2-37
2–5 How to Wake Up from Reduced Power Modes (Refer to Table 2–3 and Table 2–4) 2-38
2–6 Destination of Program Counter on Wake-Up Under Various Conditions 2-39
3–1 Interrupts 3-8
3–2 State of the Status Register (17 bit) after RESET Low-to-High (Bits 5 through 16 are left uninitialized) 3-22
4–1 Status Register (STAT) 4-7
4–2 Addressing Mode Encoding 4-9
4–3 Rx Bit Description 4-10
4–4 Addressing Mode Bits and {adrs} Field Description 4-10
4–5 MSP50P614/MSP50C614 Addressing Modes Summary 4-11
4–6 Auto Increment and Auto Decrement Modes 4-11
4–7 Flag Addressing Field {flagadrs} for Certain Flag Instructions (Class 8a) 4-12
4–8 Initial Processor State for the Examples Before Execution of Instruction 4-13
4–9 Indirect Addressing Syntax 4-15
4–10 Symbols and Explanation 4-22
4–11 Instruction Classification 4-23
4–12 Classes and Opcode Definition 4-25
4–13 Class 1 Instruction Encoding 4-26
4–14 Class 1a Instruction Description 4-26
4–15 Class 1b Instruction Description 4-27
4–16 Class 2 Instruction Encoding 4-29
4–17 Class 2a Instruction Description 4-29
4–18 Class 2b Instruction Description 4-30
4–19 Class 3 Instruction Encoding 4-31
4–20 Class 3 Instruction Description 4-31
4–21 Class 4a Instruction Encoding 4-34
4–22 Class 4a Instruction Description 4-35
4–23 Class 4b Instruction Description 4-35
4–24 Class 4c Instruction Description 4-35
4–25 Class 4d Instruction Description 4-35
4–26 Class 5 Instruction Encoding 4-36
4–27 Class 5 Instruction Description 4-36

4–28 Class 6a Instruction Encoding 4-38
4–29 Class 6a Instruction Description 4-38
4–30 Class 6b Instruction Description 4-39
4–31 Class 7 Instruction Encoding and Description 4-40
4–32 Class 8a Instruction Encoding 4-41
4–33 Class 8a Instruction Description 4-42
4–34 Class 8b Instruction Description 4-42
4–35 Class 9a Instruction Encoding 4-43
4–36 Class 9a Instruction Description 4-43
4–37 Class 9b Instruction Description 4-43
4–38 Class 9c Instruction Description 4-44
4–39 Class 9d Instruction Description 4-44
4–40 Data Memory Address and Data Relationship 4-46
4–41 MSP50P614/MSP50C614 Computational Modes 4-50
4–42 Hardware Loops in MSP50P614/MSP50C614 4-54
4–43 Initial Processor State for String Instructions 4-55
4–44 Lookup Instructions 4-57
4–45 Auto Increment and Decrement 4-73
4–46 Addressing Mode Bits and adrs Field Description 4-73
4–47 Flag Addressing Syntax and BIts 4-73
4–48 Names for cc 4-88
5–1 String Functions 5-22
7–1 Signal and Pad Descriptions for the MSP50C614 7-3
7–2 Signal and Pad Descriptions for the MSP50C605 7-4
7–3 Signal and Pad Descriptions for the MSP50C601 7-5
7–4 Signal and Pad Descriptions for the MSP50C604 7-6

Chapter 1

Introduction to the MSP50C6xx

The MSP50C6xx is a low cost, mixed signal controller, that combines a speech synthesizer, general-purpose input/output (I/O), onboard ROM, and direct speaker-drive in a single package. The computational unit utilizes a powerful new DSP which gives the MSP50C6xx unprecedented speed and computational flexibility compared with previous devices of its type. The MSP50C6xx supports a variety of speech and audio coding algorithms, providing a range of options with respect to speech duration and sound quality.

Topic **Page**

1.1	**Features of the MSP50C6xx**	**1–2**
1.2	**Applications**	**1–3**
1.3	**Development Device: MSP50P614**	**1–4**
1.4	**Functional Description for the MSP50C614**	**1–5**
1.5	**MSP50C601, MSP50C604, and MSP50C605**	**1–9**

1.1 Features of the MSP50C6xx

- Advanced, integrated speech synthesizer for high quality sound
- Operates up to 12.32 MHz (performs up to 12.32 MIPS)
- Very low-power operation, ideal for hand-held devices
- Low voltage operation, sustainable by three batteries
- Reduced power stand-by modes, less than 10 μA in deep-sleep mode
- Supports high-quality synthesis algorithms such as MELP, CELP, LPC, and ADPCM
- Contains 32K words onboard ROM (2K words reserved)
- Up to 2.36 Mbits of internal data ROM for speech storage
- 640 words RAM
- Up to 64 input/output pins
- Direct speaker driver, 32 Ω
- One-bit comparator with edge-detection interrupt service
- Resistor-trimmed oscillator or 32.768-kHz crystal reference oscillator
- Serial scan port for in-circuit emulation and diagnostics
- The MSP50C6xx is sold in die form or QFP package. An emulator device, MSP50P614 is sold in a ceramic package for development.
- **The MSP50P614 devices operate from 4.0 Vdc to 6.0 Vdc, and the MSP50C6xx devices operate from 3.0 Vdc to 5.2 Vdc**

1.2 Applications

Due to its low cost, low-power consumption, and high programmability, the MSP50C6xx is suitable for a wide variety of applications incorporating flexible I/O and high-quality speech:

- **Consumer**
  - Toys and Games
  - Appliances
  - Talking Clocks
  - Navigation Aids
- **Education**
  - Electronic Learning Aids
  - Talking Dictionaries
  - Language Translators
  - Talking Books
- **Industrial**
  - Warning Systems Controls
- **Medical**
  - Aids for the Handicapped
- **Telecom**
  - Answering Machines
  - Voice Mail Systems
- **Security**
  - Security Systems
  - Home Monitors

1.3 Development Device: MSP50P614

The MSP50P614 is an EPROM based version of the MSP50C614, and is available in a 120-pin windowed ceramic pin grid array. This EPROM based version of the device is only available in limited quantities to support software development. Since the MSP50P614 program memory is EPROM, each person doing software development should have several of these PGA packaged devices.

The MSP50P614 is also used to emulate the MSP50C601, MSP50C604, and MSP50C605 with the addition of external logic.

The MSP50C6xx code development software (EMUC6xx) supports non-real-time debugging by scanning the code sequence through the MSP50C6xx/ MSP50P614 scanport without programming the EPROM. However, the rate of code execution is limited by the speed of the PC parallel port. Any MSP50C6xx/MSP50P614 can be used in this debugging mode.

The MSP50P614 EPROM must be programmed to debug the code in real time. The EMUC6xx software is used to program the EPROM, set a breakpoint, and evaluate the internal registers after the breakpoint is reached. If a change is made to the code, the code will need to be updated and programmed into another device while erasing previous devices. This cycle of programming, debugging, and erasing typically requires 10–15 devices to be in the eraser at any one time, so 15–20 devices may be required to operate efficiently. The windowed PGA version of the MSP50P614 is required for this debugging mode.

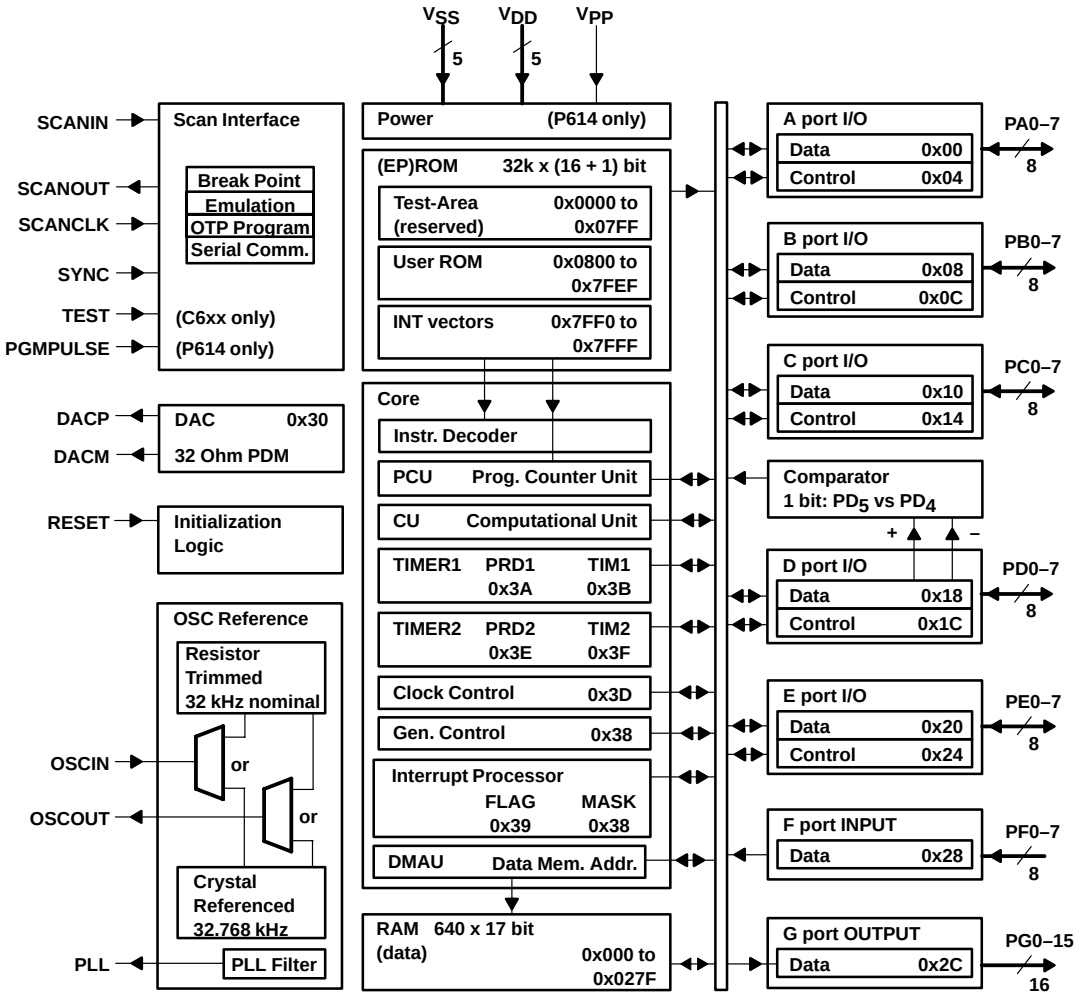
Note:

The **MSP50P614** operates with a voltage range of **4 V to 6 V**. However, the **MSP50C6xx** devices operate at a different voltage range (**3 V to 5.2 V**). Please refer to the data sheet for specific device information.

1.4 Functional Description for the MSP50C614

The MSP50C614 device consists of a micro-DSP core, embedded program and data memory, and a self-contained clock generation system. General-purpose periphery is comprised of 64 bits of flexible I/O. The block diagram appearing in Figure 1–1 gives an overview of the MSP50C614/MSP50P614 functionality.

Figure 1–1. Functional Block Diagram for the MSP50C614/MSP50P614



The core processor is a general-purpose 16 bit micro-controller with DSP capability. The basic core block includes a computational unit (CU), data address unit, program address unit, two timers, eight level interrupt processor, and several system and control registers. The core processor provides break-point capability to the MSP50C6xx code development software (EMUC6xx).

The processor is a Harvard type for efficient DSP algorithm execution. It requires separate program and data memory blocks to permit simultaneous access. The ROM has a protection scheme to prevent third-party pirating. It is configured in 32K 17-bit words.

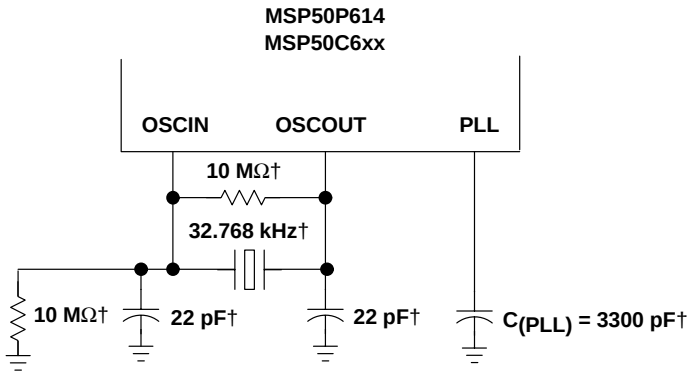
The total ROM space is divided into two areas: 1) The lower 2K words are reserved by Texas Instruments for a built-in self-test, 2) the upper 30K is for user program/data.

The data memory is internal static RAM. The RAM is configured in 640 17-bit words. Both memories are designed to consume minimum power at a given system clock and algorithm acquisition frequency.

A flexible clock generation system is included that enables the software to control the clock over a wide frequency range. The implementation uses a phase-locked loop (PLL) circuit to generate the processor clock. The Processor clock is programmable in 65.536-kHz steps between 64 kHz and 12.32 MHz. The PLL reference clock is also selectable; either a resistor-trimmed oscillator or a crystal-referenced oscillator may be used. Internal and peripheral clock sources are controlled separately to provide different levels of power management (see Figure 1–2).

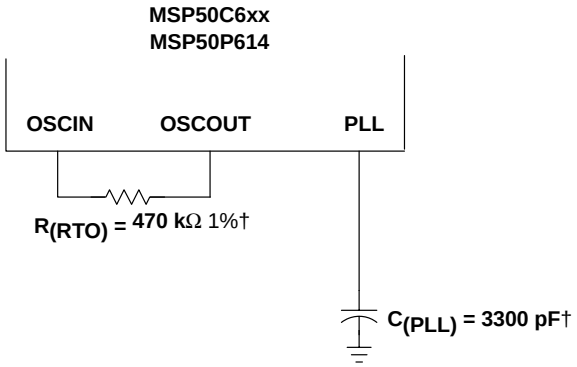
Figure 1–2. Oscillator and PLL Connection

a) Crystal Reference Oscillator Connections



† Keep these components as close as possible to the OSC_{IN}, OSC_{OUT}, and PLL pins.

b) Resistor Trim Oscillator Connections



† Keep these components as close as possible to the OSC_{IN}, OSC_{OUT}, and PLL pins.

The peripheral consists of five 8-bit wide general-purpose I/O ports, one 8-bit wide dedicated input port, and one 16-bit wide dedicated output port. The general-purpose I/O ports are bit-wise programmable as either high-impedance inputs or as totem-pole outputs. They are controlled via addressable I/O registers. The input-only port has a programmable pullup option (100-kΩ minimum resistance) and a dedicated service interrupt. These features make the input port especially useful as a key-scan interface.

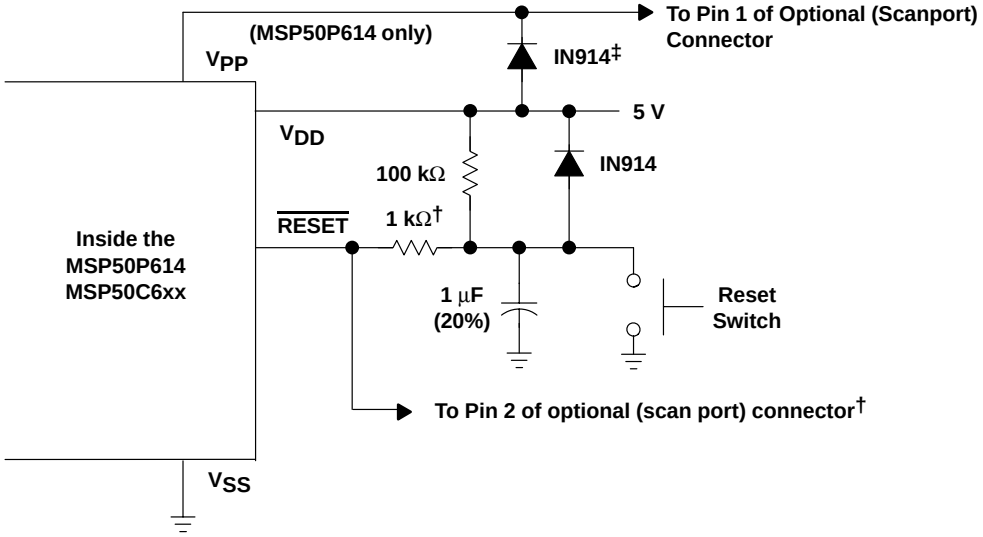
A simple one-bit comparator is also included in the periphery. The comparator is enabled by a control register, and its input pins are shared with two pins in one of the general-purpose I/O ports.

Rounding out the MSP50C6xx periphery is a built in pulse-density-modulated (PDM) digital-to-analog converter (DAC) with direct speaker-drive capability.

Typical connections to implement reset functionality are shown in Figure 1–3.

An external reset circuit is required to hold the reset pin low until the MSP50C6xx power supply has stabilized in the specified voltage range. In some cases, a simple reset circuit (as shown in Figure 1–3) can be used for this purpose. However, this simple circuit may not be suitable for all applications. For example, if the power supply has an unpredictable rise time or has intermittent voltage sags, the device may not initialize properly. The diode and the switch shown in Figure 1–3 may be optional for some applications. The diode provides a lower impedance path for the capacitor to discharge when power is removed. This make the circuit more reliable when power is removed and quickly reapplied.

Figure 1–3. RESET Circuit



† If it is necessary to use the software development tools to control the MSP50P614 in an application board, the 1 kΩ resistor is needed to allow the development tool to over drive the RESET circuit on the application board.

‡ This Diode can be omitted (shorted) if the application does not require use of the scanport interface. See Section 7.1 regarding scan port bond out.

Note:

This simple circuit may not be suitable for all applications. For example, if the power supply has an unpredictable rise time or has intermittent voltage sags, the device may not initialize properly.

1.5 MSP50C601, MSP50C604, and MSP50C605

Related products, the MSP50C601, MSP50C604, and MSP50C605 use the MSP50C6xx core. The MSP50C601 has a 128K byte data ROM built into the chip and 32 I/O port pins. The MSP50C605 has a 224K byte data ROM built into the chip and 32 I/O port pins. The MSP50C604 has a 64K byte data ROM built into the chip and 16 I/O port pins. The MSP50C601 can provide up to 24 minutes, the MSP50C605 can provide up to 37 minutes, and the MSP50C604 can provide up to 6.5 minutes of uninterrupted speech. The MSP50C604 is designed to support slave operation with an external host microcontroller. In this mode the MSP50C604 can be programmed with a code that communicates with the host via a command set. This command set can be designed to support LPC, CELP, MELP, and ADPCM coders by selecting the appropriate command. The MSP50C604 can also be used stand-alone in master mode. The MSP50C601, MSP50C604, and MSP50C605 use the MSP50P614 as the development version device.

Chapter 2

MSP50C6xx Architecture

A detailed description of the MSP50C6xx architecture is included in this chapter. After reading this chapter, the reader will have in-depth knowledge of internal blocks, memory organization, interrupt system, timers, clock control mechanism, and various low power modes.

Topic **Page**

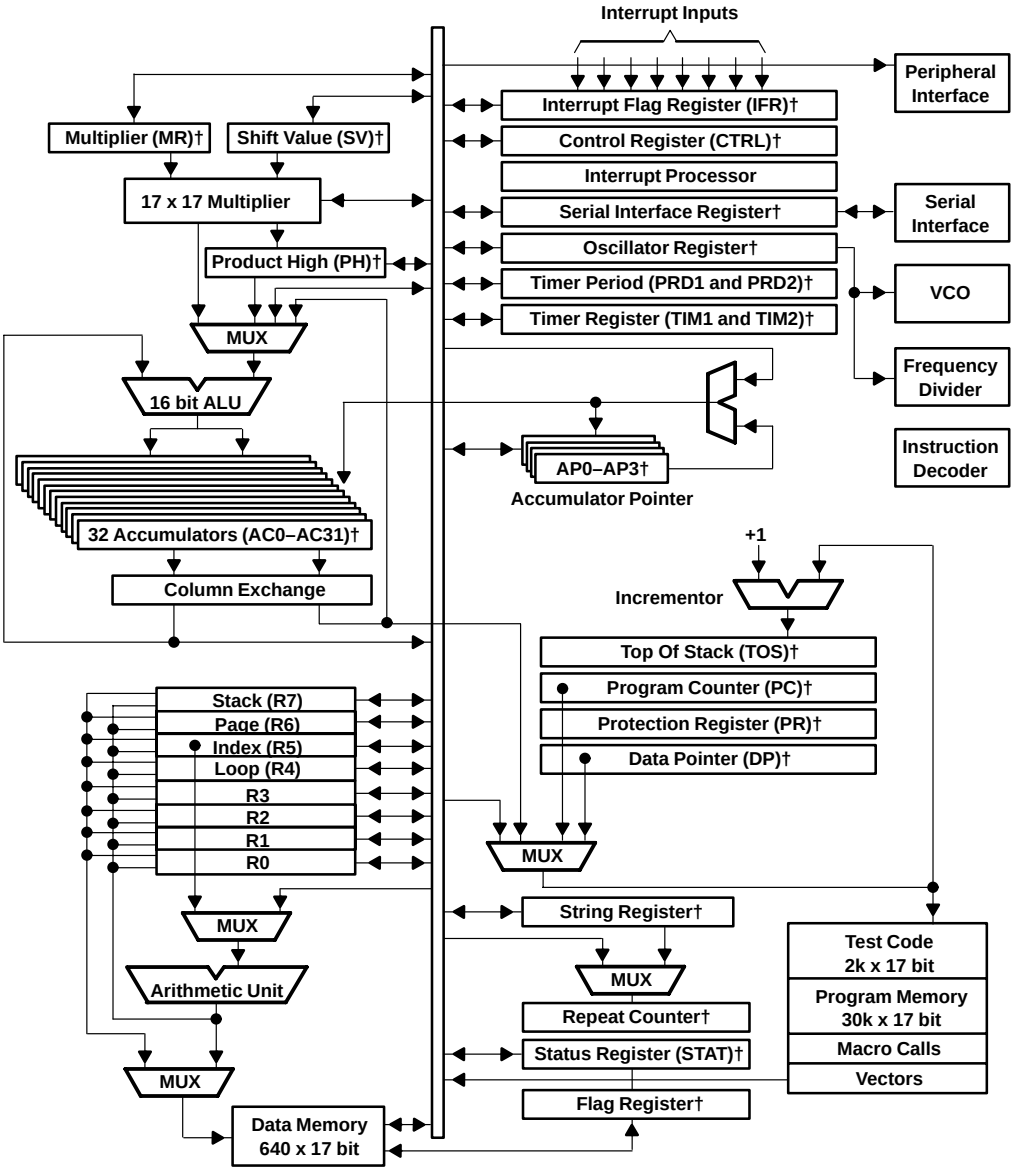
2.1	Architecture Overview	2–2
2.2	Computation Unit	2–5
2.3	Data Memory Address Unit	2–11
2.4	Program Counter Unit	2–14
2.5	Bit Logic Unit	2–14
2.6	Memory Organization: RAM and ROM	2–15
2.7	Interrupt Logic	2–22
2.8	Clock Control	2–26
2.9	Timer Registers	2–31
2.10	Reduced Power Modes	2–33
2.11	Execution Timing	2–40

2.1 Architecture Overview

The core processor in the C6xx is a medium performance mixed signal processor with enhanced microcontroller features and a limited DSP instruction set. In addition to its basic multiply/accumulate structure for DSP routines, the core provides for a very efficient handling of string and bit manipulation. A unique accumulator-register file provides additional scratch pad memory and minimizes memory thrashing for many operations. Five different addressing modes and many short direct references provide enhanced execution and code efficiency.

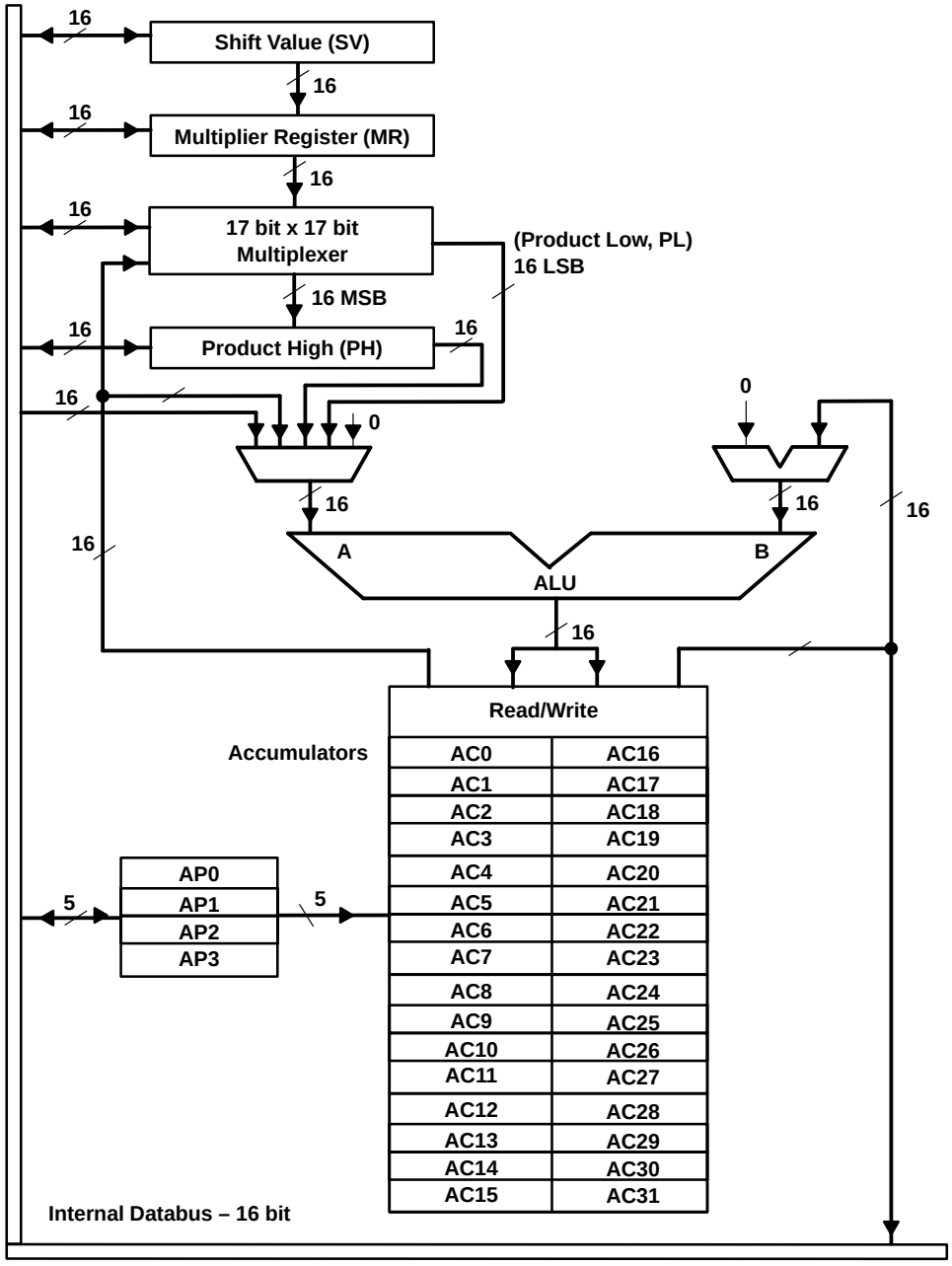
The basic elements of the C6xx core are shown in Figure 2–1. In addition to the main computational units, the core's auxiliary functions include two timers, an eight-level interrupt processor, a clock generation circuit, a serial scan-port interface, and a general control register.

Figure 2–1. MSP50C6xx Core Processor Block Diagram



† Indicates internal programmable registers.

Figure 2–2. Computational Unit Block Diagram



2.2 Computation Unit

The computational unit (CU) is comprised of a (17-bit by 17-bit) Booth's algorithm multiplier and a 16-bit arithmetic logic unit (ALU). The block diagram of the CU is shown in Figure 2–2. The multiplier block is served by 4 system registers: a 16-bit multiplier register (MR), a 16-bit write-only multiplicand register, a 16-bit high word product register (PH), and a 4-bit shift value register (SV). The output of the ALU is stored in one 16-bit accumulator from among the 32 which compose the accumulator-register block. The accumulator register block can supply either one operand to the ALU (addressed accumulator register or its offset register) or two operands to the ALU (both the addressed register and its offset).

2.2.1 Multiplier

The multiplier executes a 17-bit by 17-bit 2s complement multiply and multiply-accumulate in a single instruction cycle. The sign bit within each operand is bit 16, and its value extends from bit 0 (LSB) to bit 15 (MSB). The sign bit for either operand (multiplier or multiplicand) can assume a positive value (zero) or a value equal to the MSB (bit 15). In assuming zero, the extra bit supports unsigned multiplication. In assuming the value of bit 15, the extra bit supports signed multiplication. Table 2–1 shows the greater magnitude achievable when using unsigned multiplication (65535 as opposed to 32767).

Table 2–1. Signed and Unsigned Integer Representation

Unsigned		Signed	
Decimal	**Hex**	**Decimal**	**Hex**
65535	0xFFFF	–1	0xFFFF
32768	0x8000	–32768	0x8000
32767	0x7FFF	32767	0x7FFF
0	0x0000	0	0x0000

During multiplication, the lower word (LSB) of the resulting product, product low, is multiplexed to the ALU. Product low is either loaded to or arithmetically combined with an accumulator register. These steps are performed within the same instruction cycle. Refer to Figure 2–3 for an overview of this operation. At the end of the current execution cycle, the upper word (MSB) of the product is latched into the product high register (PH).

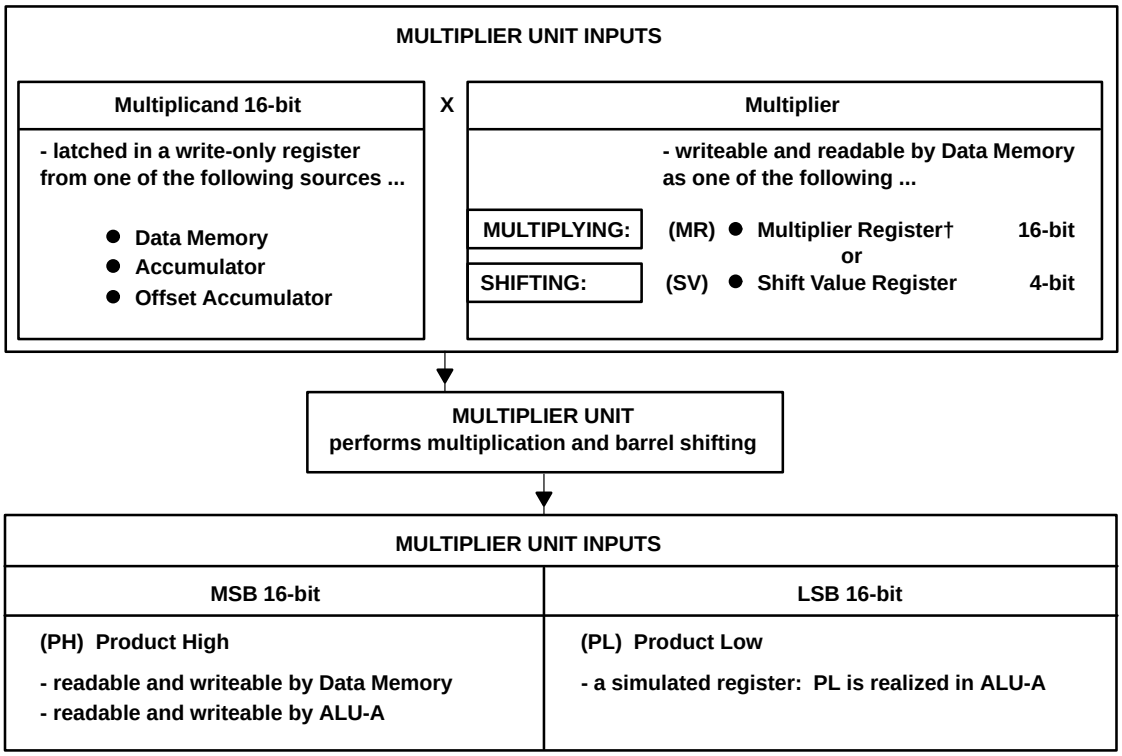
The multiplicand source can be either data memory, an accumulator, or an accumulator offset. The multiplier source can be either the 16-bit multiplier register (MR) or the 4-bit shift value (SV) register. For all multiply operations, the MR register stores the multiplier operand. For barrel shift instructions, the multiplier operand is a 4-to-16-bit value that is decoded from the 4-bit shift value register (SV).

As an example of a barrel shift operation, a coded value of 0x7 in the SV register results in a multiplier operand of 0000000010000000 (1 at bit 7). This causes a left-shift 7-times on the 16 bit multiplicand. The output result is 32-bit. On the other hand, if the status bit FM (multiplier shift mode) is SET, then the multiplier operand (0000000010000000) is left-shifted once to form a 17 significant-bit operand (00000000100000000). This mode is included to avoid a divide-by-2 of the product, when interpreting the input operands as signed binary fractions. The multiplier shift mode status bit is located in the status register (STAT).

All three multiplier registers (PH, SV, and MR) can be loaded from data memory and stored to data memory. In addition, data can be transferred from an accumulator register to the PH, or vice versa. Both long and short constants can be directly loaded to the MR from program memory.

The multiplicand is latched in a write-only register from the internal data bus. The value is not accessible by memory or other system registers.

Figure 2–3. Overview of the Multiplier Unit Operation



† Also write-able by Program Memory

2.2.2 Arithmetic Logic Unit

The arithmetic logic unit is the focal point of the computational unit, where data can be added, subtracted, and compared. Logical operations can also be performed by the ALU. The basic hardware word-length of the ALU is 16 bits; however, most ALU instructions can also operate on strings of 16-bit words (i.e., a series or array of values). The ALU operates in conjunction with a flexible, 16-bit accumulator register block. The accumulator register block is composed of thirty-two, 16-bit registers which further enhances execution and promotes compact code.

The ALU has two distinct input paths, denoted ALU-A and ALU-B (see Figure-2–4). The ALU-A input selects between all zeros, the internal databus, the product high register (PH), the product low (PL), or the offset output of the accumulator register block. The ALU-B input selects between all zeros and the output from the accumulator register block.

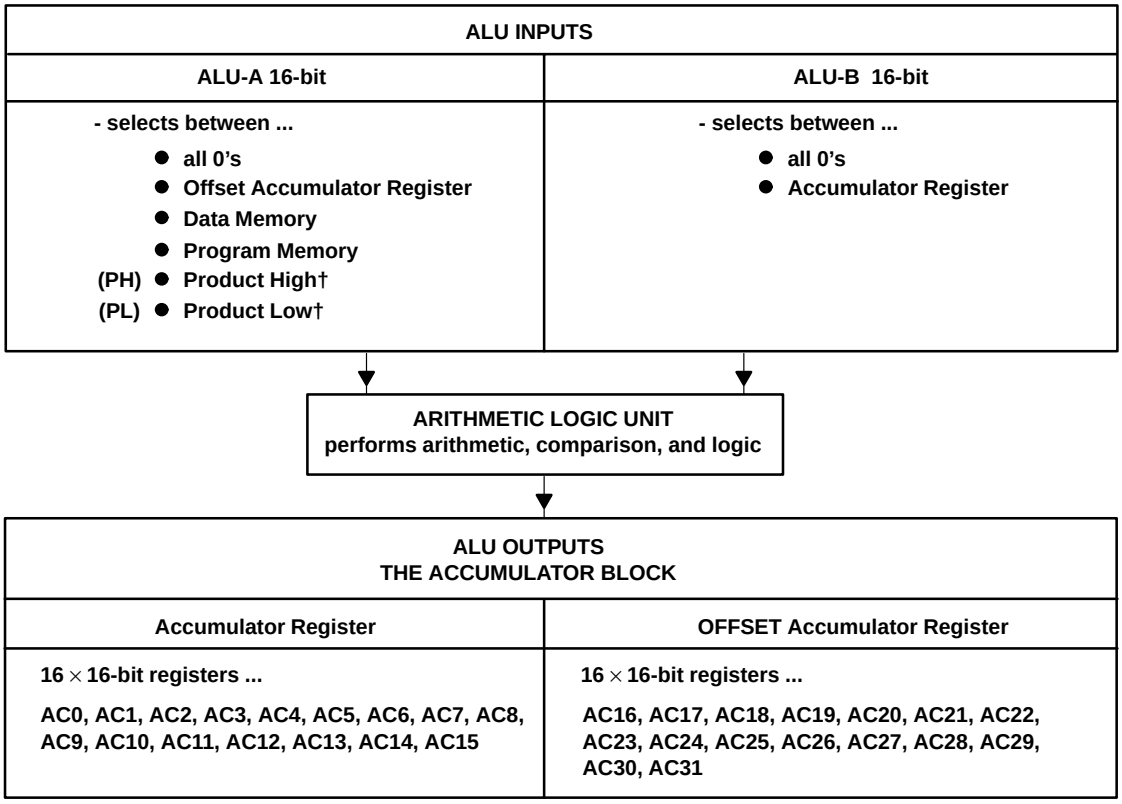
The all-zero values are necessary for data transfers and unitary operations. All-zeros also serve as default values for the registers, which helps to minimize residual power consumption. The databus path through ALU-A is used to input memory values (RAM) and constant values (program memory) to the ALU. The PH and PL inputs are useful for supporting multiply-accumulate operations (refer to Section 2.2.1, *Multiplier*).

The operations supported by the ALU include arithmetic, logic, and comparison. The arithmetic operations are addition, subtraction, and load (add to zero). The logical operations are AND, OR, XOR, and NOT. Comparison includes equal-to and not-equal-to. The compare operations may be used with constant, memory, or string values without destroying any accumulator values.

2.2.2.1 Accumulator Block

The output of the ALU is the accumulator block. The accumulator block is composed of thirty-two, 16-bit registers. These registers are organized into two terminals, denoted accumulator and OFFSET accumulator. The terminals provide references for all of the data which is to be held in the accumulator block. The accumulator incorporates one-half of the 32 accumulator registers: AC0..AC15. The OFFSET accumulator incorporates the other half: AC16..AC31.

Figure 2–4. Overview of the Arithmetic Logic Unit



† For multiply-accumulate operations.

2.2.2.2 Accumulator Pointer Block

There are four 5-bit registers which are used to store pointers to members of the accumulator block. The accumulator pointers (AP0, AP1, AP2, AP3) are used in two modes: 1) as a direct reference to one of 32, or 2) as an indirect reference. The indirect reference includes a direct reference to one of 16 and an offset (optional) which increments the reference by 16: AC(N+16). For example, AC0 has its offset register located at AC16. AC1 has an offset register located at AC17, and so on. The block is circular: address 31, when incremented, results in address 0. The offsets of AC16 through AC31, therefore, are AC0 through AC15, respectively (see Figure 2–5). Indirect referencing by the AP pointers is supported by most of the C6xx's accumulator-referenced instructions.

When writing an accumulator-referenced instruction, therefore, the working accumulator address is stored in one of AP0 to AP3. The C6xx instruction set provides a two-bit field for all accumulator referenced instructions. The two-bit field serves as a reference to the accumulator pointer which, in turn, stores the address of the actual 16-bit accumulator. Some MOV instructions store the contents of the APn directly to memory or load from memory to the APn register. Other instructions can add or load 5-bit constants to the current APn register contents. A full description of the C6xx instruction set is given in Chapter 4, *Assembly Language Instructions*.

Figure 2–5. Overview of the Accumulators

Accumulator Block:	32, 16-bit registers	**AC(0) . . . AC(31)**
Accumulator Block Pointers:	4, 5-bit registers	**AP(0) . . . AP(3)**

The accumulator block pointers may assume values in one of two forms:

1) DIRECT REFERENCE:	**0 . . . 31**		
			AC Register #
2) INDIRECT REFERENCE:	**0 . . . 15**	points to:	**0 . . . 15**
	0 . . . 15 OFFSET	points to:	**16 . . . 31**
	15 . . . 31 OFFSET	points to:	**0 . . . 15**

– AP registers are served by a 5-bit processor for sequencing addresses or repetitive operations.

– Selection between the 4 AP's is made in the 2-bit An field in all accumulator-referenced instructions

2.2.2.3 String Operations

The AP registers are served by a 5-bit processor that provides efficient sequencing of accumulator addresses. The design automates repetitive operations like long data strings or repeated operations on a list of data.

When operating on a multiword data string, the address is copied from the AP register to fetch the least significant word of the string. This copy is then consecutively incremented to fetch the next n words of the string. At the completion of the consecutive operations, the actual address stored in the AP register is left unchanged; its value still points to the least significant location. The AP register, therefore, is loaded and ready for the **next** repeatable operation.

For some instructions, the 5-bit string processor can also preincrement or predecrement the AP pointer-value by +1 or –1, before being used by the accumulator register block. This utility can be effectively used to minimize software overhead in manipulating the accumulator address. The premodification of the address avoids the software pipelining effect that post-modification would cause.

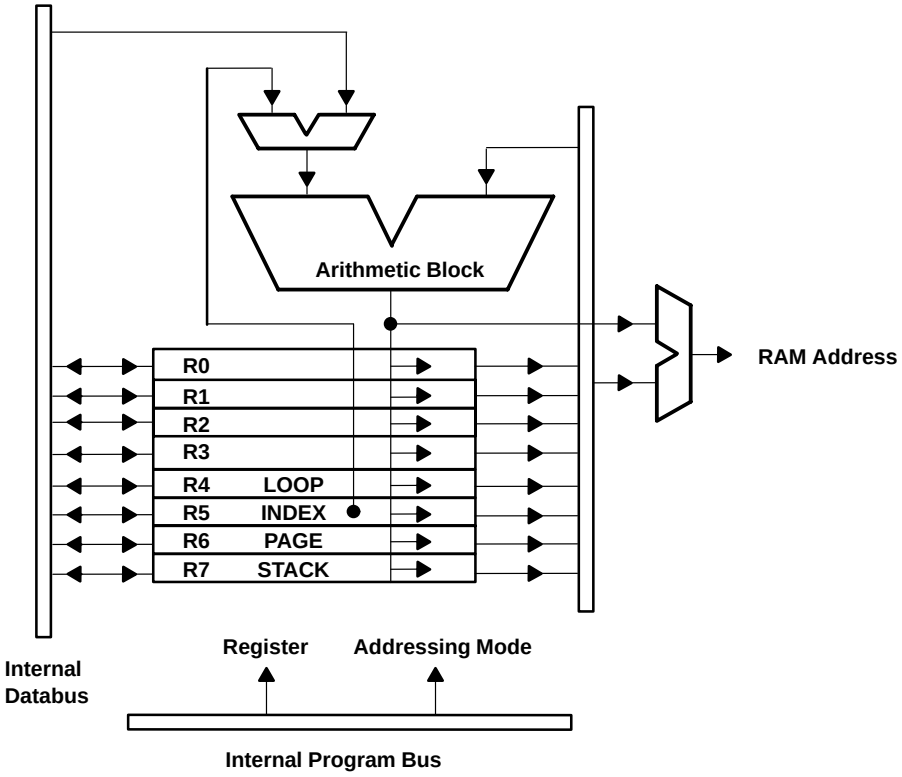
Some C6xx instructions reference only the accumulator register and cannot use or modify the offset register that is fetched at the same time. Other instructions provide a selection field in the instruction word (A~ or ~A op-code bit). This has the effect of exchanging the column addressing sense and thus the source or order of the two registers. Also, some instructions can direct the ALU output to be written either to the accumulator register or to the offset accumulator register. Refer to Chapter 4, *Assembly Language Instructions*, for more details.

The ALU's accumulator block functions as a small workspace, which eliminates the need for many intermediate transfers to and from memory. This alleviates the memory thrashing which frequently occurs with single accumulator designs.

2.3 Data Memory Address Unit

The data memory address unit (DMAU) provides addressing for data memory (internal RAM). The block diagram of the DMAU is shown in Figure 2–6. The unit consists of a dedicated arithmetic block and eight read/write registers (R0 through R7). Each read/write register is 16-bits in size. The arithmetic block is used to add, subtract, and compare memory-address operands. The register set includes four general-purpose registers (R0 to R3) and four special-purpose registers. The special-purpose registers are: the LOOP control register (R4), the INDEX register (R5), the PAGE register (R6), and the STACK register (R7). The DMAU generates a RAM address as output. The DMAU functions completely in parallel with the computational unit, which helps the C6xx maintain a high computational throughput.

Figure 2–6. Data Memory Address Unit



2.3.1 RAM Configuration

The data memory block (RAM) is physically organized into 17-bit parallel words. Within each word, the extra bit (bit 16) is used as a flag bit or tag for op-codes in the instruction set. Specifically, the flag bit directs complex branch conditions associated with certain instructions. The flag bit is also used by the computational unit for signed or unsigned arithmetic operations (see Section 2.2.1, *Multiplier*).

The size of the C6xx RAM block is 640 17-bit locations. Each address provided by the DMAU causes 17 bits of data to be addressed. These 17 bits are operated on in different ways, depending on the instructions being executed. For most instructions, the data is interpreted as 16-bit word format. This means that bits 0 through 15 are used, and bit 16 is either ignored or designated as a flag or status bit.

There are two-byte instructions, for example MOVB, which cause the processor to read or write data in a byte (8-bit) format. (The B appearing at the end of MOVB designates it as an instruction that uses byte-addressable arguments.) The byte-addressable mode causes the hardware to read/write either the upper or lower 8 bits of the 16-bit word based on the LSB of the address. In this case, the address is a byte address, rather than a word address. Bits 0 through 7 within the word are used, so that a single byte is automatically right-justified within the databus. Bits 8 through 15 may also be accessed as the upper byte at that same address.

A third data-addressing mode is the flag data mode, whereby, the instruction operates on only the single flag bit (bit 16) at a given address. All flag mode instructions execute in one instruction cycle. The flags can be referenced in one of two addressing modes: 1) global address, whereby 64 global flags are located at fixed locations in the first 64 RAM addresses, and 2) flag relative address, whereby a reference is made relative to the current PAGE (R6). The relative address supports 64 different flags whose PAGE-offset values are stored in the PAGE register. The flag mode instructions cannot address memory in the INDEX-relative modes. See Chapter 4, *Assembly Language Instructions*, for more details.

2.3.2 Data Memory Addressing Modes

The DMAU provides a powerful set of addressing modes to enhance the performance and flexibility of the C6xx core processor. The addressing modes for RAM fall into three categories:

- ❏ Direct addressing
- ❏ Indirect addressing with post-modification
- ❏ Relative addressing

The relative addressing modes appear in three varieties:

- ❏ Immediate Short, relative to the PAGE (R6) register.
  The effective RAM address is: [*R6 + (a 7 bit direct offset)].

- ❏ Relative to the INDEX (R5) register.
  The effective RAM address is: [*R5 + (an indexed offset)].

- ❏ Long Immediate, relative to the register base.
  The effective RAM address is: [*R*x* + (a 16 bit direct offset)].

Refer to Chapter 4, *Assembly Language Instructions*, for a full description of how these modes are used in conjunction with various instructions.

2.4 Program Counter Unit

The program counter unit provides addressing for program memory (onboard ROM). It includes a 16-bit arithmetic block for incrementing and loading addresses. It also consists of the program counter (PC), the data pointer (DP), a buffer register, a code protection write-only register, and a hardware loop counter (for strings and repeated-instruction loops). The program counter unit generates a ROM address as output.

The program counter value, PC, is automatically saved to the stack on various CALL instructions and interrupt service branches. The stack consists of one hardware-level register (TOS) which points to the top-of-stack. The TOS is followed by a software stack. The software stack resides in RAM and is addressed using the STACK register (R7) in indirect mode (see Section 2.3, *Data Memory Address Unit*).

The hardware loop counter controls the execution of repeated instructions using one of two modes: 1) consecutive iterations of a single instruction following the repeat (RPT) instruction, or 2) a single instruction which operates on a string of data values (string loops). For all types of repeated execution, interrupt service branches are automatically disabled (temporarily).

The data pointer (DP) register is loaded at two instances: 1) from the accumulator during lookup-table instructions, and 2) from the databus during the fetch of long string constants. To simplify algorithms which require sequential indices to lookup tables, the DP register may be stored in RAM.

2.5 Bit Logic Unit

The bit logic unit is a 1-bit unit which operates on flag bit data. It is controllable by eleven different instructions, which generate the decision flags for conditional program control. The results of operations performed by the bit logic unit are sent either to the flag bit of RAM memory or to the TF1 and TF2 bits of the status register (STAT).

2.6 Memory Organization: RAM and ROM

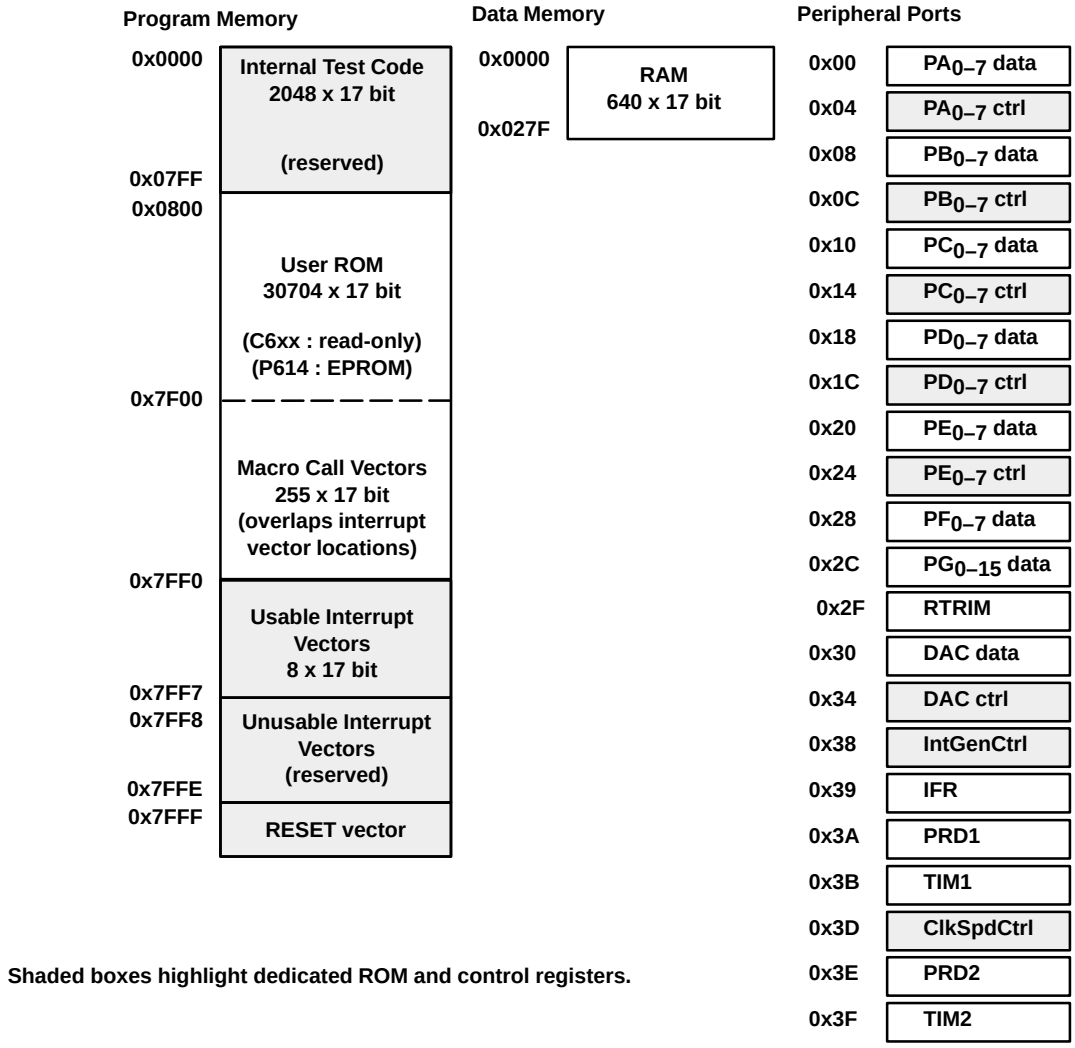
Data memory (RAM) and program memory (ROM) are each restricted to internal blocks on the C6xx. The program memory is read-only and limited to 32K, 17-bit words. The lower 2048 of these words is reserved for an internal test code and is not available to the user. The data memory is static RAM and is limited to 640, 17-bit words. 16 bits of the 17-bit RAM are used for the data value, while the extra bit is used as a status flag.

The C6xx does not have the capability to execute instructions directly from external memory. However, additional program memory (external ROM) can be accessed using the general-purpose I/O. The interface for external ROM must be configured in the software.

2.6.1 Memory Map

The memory map for the C6xx is shown in Figure 2–7. Refer to Section 2.6.3, *Interrupt Vectors*, for more detailed information regarding the interrupt vectors, and to Section 2.6.2, *Peripheral Communications (Ports)*, for more information on the I/O communications ports.

Figure 2–7. C6xx Memory Map (not drawn to scale)



Shaded boxes highlight dedicated ROM and control registers.

2.6.2 Peripheral Communications (Ports)

Peripheral functions in the C6xx are controlled using one or more of the I/O address-mapped communications ports. Table 2–2 describes the ports.

The width of each mapped location, shown in width of location, is independent of the address spacing. In other words, some registers are smaller in width than the spacing between neighboring addresses. The few unused bits appear to the right of the LSB values within the DAC Data register, address 0x30 (refer to Section 3.2.2, *DAC Control and Data Registers*).

When writing to any of the locations in the I/O address map, therefore, the bit-masking need only extend as far as width of location. Within a 16-bit accumulator, the desired bits (width of location) should be right-justified. The write operation is accomplished using the OUT instruction, with the address of the I/O port as an argument.

A read from these locations is accomplished using the IN instruction, with the address of the I/O port as an argument. When reading from the I/O port to a 16-bit accumulator, the IN instruction automatically clears any extra bits in excess of width of location. The desired bits in the result will be right-justified within the accumulator.

Allowable access indicates whether the port is bidirectional, read-only, or write-only. The last column of the table points to the section in this manual where the functions of each bit have been defined in more detail.

Table 2–2. Summary of MSP50C614's Peripheral Communications Ports

I/O Map Address	Width of Location	Allowable Access	Control Register Name	Abbreviation	State after RESET LOW	Section for Reference
0x00	8 bits	Read & Write	I/O port A data	$PA_{0..7}$ Data	unknown†	3.1.1
0x04	8 bits	Read & Write	I/O port A control	$PA_{0..7}$ Ctrl	0x00 ‡	
0x08	8 bits	Read & Write	I/O port B data	$PB_{0..7}$ Data	unknown	
0x0C	8 bits	Read & Write	I/O port B control	$PB_{0..7}$ Ctrl	0x00	
0x10	8 bits	Read & Write	I/O port C data	$PC_{0..7}$ Data	unknown	
0x14	8 bits	Read & Write	I/O port C control	$PC_{0..7}$ Ctrl	0x00	
0x18	8 bits	Read & Write	I/O port D data	$PD_{0..7}$ Data	unknown	
0x1C	8 bits	Read & Write	I/O port D control	$PD_{0..7}$ Ctrl	0x00	
0x20	8 bits	Read & Write	I/O port E data	$PE_{0..7}$ Data	unknown	
0x24	8 bits	Read & Write	I/O port E control	$PE_{0..7}$ Ctrl	0x00	
0x28	8 bits	Read Only	Input port F data	$PF_{0..7}$ Data	unknown	3.1.2
0x2C	16 bits	Read & Write	Output port G data	$PG_{0..15}$ Data	0x0000	3.1.3
0x2F	17 bits	Read Only	RTO oscillator trim adjustment	RTRIM	0x0000	2.8.4
0x30	16 bits	Write Only	DAC data	DAC Data	0x0000	3.2.2
0x34	4 bits	Read & Write	DAC control	DAC Ctrl	0x0	3.2.2
0x38	16 bits	Read & Write	Interrupt/general Ctrl	IntGenCtrl	0x0000	3.4

† Input states are provided by the external hardware.
‡ A control register value of 0x00 yields a port configuration of all inputs.

Table 2–2. Summary of C614's Peripheral Communications Ports (Continued)

I/O Map Address	Width of Location	Allowable Access	Control Register Name	Abbreviation	State after RESET LOW	Section for Reference
0x39	8 bits	Read & Write	Interrupt flag	IFR	Same state as before RESET	2.7
0x3A	16 bits	Read & Write	TIMER1 period	PRD1	0x0000	2.8
0x3B	16 bits	Read & Write	TIMER1 count-down	TIM1	0x0000	
0x3D	16 bits	Write Only	Clock speed control	ClkSpdCtrl	0x0000	2.9.3
0x3E	16 bits	Read & Write	TIMER2 period	PRD2	0x0000	2.8
0x3F	16 bits	Read & Write	TIMER2 count-down	TIM2	0x0000	

2.6.3 Interrupt Vectors

When its event has triggered and its service has been enabled, an interrupt causes the program counter to branch to a specific location. The destination location is stored (programmed) in the interrupt vector, which resides in an upper address of ROM. The following table lists the ROM address associated with each interrupt vector:

Interrupt Name	ROM address of Vector	Event Source	Interrupt Priority
INT0	0x7FF0	DAC Timer	Highest
INT1	0x7FF1	TIMER1	2nd
INT2	0x7FF2	TIMER2	3rd
INT3	0x7FF3	port D_2	4th
INT4	0x7FF4	port D_3	5th
INT5	0x7FF5	all port F	6th
INT6	0x7FF6	port D_4	7th
INT7	0x7FF7	port D_5	Lowest
	0x7FFE	storage for ROM Protection Word	
RESET	0x7FFF	storage for initialization vector	

Note: ROM Locations that Hold Interrupt Vectors

ROM locations that hold interrupt vectors are reserved specifically for this purpose. Additional ROM locations 0x7FF8 - 0x7FFD are reserved for future expansion. Like the interrupt vectors, they should not be used for general program storage.

The branch to the program location that is specified in the interrupt vector is, of course, contingent on the occurrence of the trigger event. Refer to Section 3.1.5, *Internal and External Interrupts*, for more information regarding the specific conditions for each interrupt-trigger event. The branch operation, however, is also contingent on whether the interrupt service has been enabled. This is done individually for each interrupt, using the interrupt mask bits within the interrupt/general control register. Refer to Section 2.7, *Interrupt Logic*, for more details.

The ROM location 0x7FFF holds the program destination associated with the hardware RESET event (branch happens after RESET LOW-to-HIGH). The location 0x7FFE holds the read/write block-protection word. Refer to Section 2.6.4, *ROM Code Security*, for an explanation of the ROM security scheme.

2.6.4 ROM Code Security

The C6xx provides a mechanism for protecting its internal ROM code from third-party pirating. The protection scheme is composed of two levels, both of which prevent the ROM contents from being read. Protection may be applied to the entire program memory, or it can be applied to a block of memory beginning at address 0x0000 and ending at an arbitrary address. The two levels of ROM protection are designated as follows:

- Direct read and write protection, via the ROM scan circuit.
- Indirect read protection, which prohibits the execution of memory-lookup instructions.

For the purposes of direct security, the ROM is divided into two blocks. The first block begins at location 0x0000, and ends, inclusively, at location ($m \times 512 - 1$), where m is some integer. Each address specifies a 17-bit word location. The second block begins at location ($m \times 512$), and ends, inclusively, at 0x7FFF (the end of the ROM). The first block is protected from reads and writes by programming a block protection bit, and the second block is protected from reads and writes by programming a global protection bit.

The two-block system is designed in such a way that a secondary developer is prevented from changing the partition address between blocks. Once the block protection has been engaged, then the only security option available to the secondary developer is engaging the global protection.

Note: Instructions with References

Care must be taken when employing instructions that have either long string constant references or look-up table references. These instructions will execute properly only if the address of the instruction and the address of the data reference are within the same block.

The protection modes are implemented on the C6xx as follows. Within the ROM is a dedicated storage for the block protection word (address 0x7FFE). The block protection word is divided into two 6-bit fields and two single-bit fields. The remainder of the 17-bit word is broken into three single-bit fields which are reserved for future use.

Block Protection Word

address 0x7FFE (17-bit wide location)

	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
WRITE only	R	R	TM	TM	TM	TM	TM	TM	GP	BP	R	FM	FM	FM	FM	FM	FM
			05	04	03	02	01	00				05	04	03	02	01	00

TM : True Protection Marker **(N_{TM})**

FM : False Protection Marker **(N_{FM})**

R : Reserved for future use (must be 1)

GP : Global Protection (0 value protects)

BP : Block Protection (0 value protects)

1 : Default value of cells on erasure

The two 6-bit fields are designated as the true protection marker, (TM5 through TM0) and the false protection marker, (FM5 through FM0). When setting up a partition for partial ROM protection, the address of the partition must be specified as:

$[(N_{TM} + 1) * 512 – 1]$	= highest ROM address within the block to be protected
$(N_{TM} + 1) * 512$	= lowest ROM address which is left unprotected
N_{TM}	= the value programmed at TM5…TM0 (true protection marker)
N_{FM}	≡ the binary complement of N_{TM}
N_{FM}	= the value programmed at FM5…FM0 (false protection marker)

The purpose of the true and false protection markers is to provide parity. An erased P614 EPROM cell defaults to the value 1. Once programmed from 1 to 0, it cannot be programmed back to 1, unless the cell (and all other cells along with it) are subject to erasure. A multi-pass programming, therefore, can only lower the value stored at an EPROM address and never raise it. Once a valid block-partition address has been properly specified in both TM and FM, it is impossible to change TM to another address and still maintain parity with FM.

Note: Block Protection Mode

When applying the block protection mode, bits FM5 through FM0 **must** be programmed as the logical inverse of bits TM5 through TM0, respectively.

Across the span of the 32k word ROM space, there are 64 possible values for N_{TM} (including zero). Hence, the 6-bit-wide locations for TM and FM.

The two single-bit fields found within the block protection word are the block protection bit (BP) and the global protection bit (GP). If BP and GP are both SET (erased), then no protection is applied to the ROM.

If BP is CLEAR and GP is SET, then the block protection mode is engaged. This means that read and write access is prevented at locations 0x0000 through $[(N_{TM} + 1) \times 512 – 1]$. Read and write access is permitted at locations $[(N_{TM} + 1) \times 512]$ through 0x7FFF.

If GP is CLEAR, then the global protection mode is engaged. This prevents read and write access to all addresses of the ROM, regardless of the value of BP.

Note: Block Protection Word

The remaining bits in the block protection word are reserved for future use, but must remain **set** in order to ensure future compatibility. These bits are numbers 6, 15, and 16.

When the device is powered up, the hardware initialization circuit reads the value stored in the block protection word. The value is then loaded to an internal register and the security state of the ROM is identified. Until this occurs, execution of any instructions is suspended.

The same initialization sequence is executed before entry into the special test-modes available on the P614 and C6xx (EPROM mode, emulation mode, and trace mode). This insures that the protection scheme is always in force when running the processor in one of these modes. A dedicated circuit ensures that a switch between emulation mode and trace mode cannot occur without going through the initialization (security check). This forces all look-up tables and long constant references to originate from an external program source, when in emulation mode. It is possible to switch from trace mode to emulation mode by lowering V_{PP}, but this transition, by design, does **not** jeopardize code security.

2.6.5 Macro Call Vectors

Macro call vectors are similar to CALL instructions except they take an 8-bit address. The upper 8 bits is always 7Fh. See Section 4.14.84, *VCALL*, for more information on the VCALL instruction.

2.7 Interrupt Logic

An eight-level interrupt system is included as part of the C6xx's core processor. The initialization and control of these interrupts is governed by the following components: the global interrupt enable, the interrupt flag register, the interrupt mask register, and the interrupt service branch. Each of these is described below.

Interrupts must be globally enabled using the INTE instruction, and they are globally disabled using the INTD instruction. INTE sets the global interrupt enable bit, and INTD clears the global interrupt enable bit. The state of this bit specifically determines whether any interrupt service branches will be taken. The global interrupt enable appears as bit 4 within the status register (STAT).

Note:

To ensure proper executions of the INTD instruction, it is recommended that the INTD instruction be prescaled with a RPT 2–2 instruction.

Each interrupt level waits for the conditions of its trigger event (refer to Figure 2–8). At the time that a trigger event occurs, the respective bit is

automatically SET in the interrupt flag register (IFR). The IFR is an 8-bit wide port-addressed register; wherein, each interrupt level is represented. A set bit in the IFR indicates that the interrupt is pending and waiting to be serviced. A clear bit indicates that the interrupt is not currently pending. The address of the IFR is 0x39. After a RESET low, the IFR is left in the same state it was before the RESET low, assuming there is no interruption in power. For a full description of the interrupt-trigger events, refer to Section 3.1.5, *Internal and External Interrupts*.

(8-bit wide location)

	07	06	05	04	03	02	01	00	← INT number
IFR Interrupt Flag register address 0x39	D5 low priority	D4	PF	D3	D2	T2	T1	DA high priority	

D5 : port D_5 falling-edge†

D4 : port D_4 rising-edge†

D3 : port D_3 falling-edge

D2 : port D_2 rising-edge

PF : any port **F** falling-edge

T2 : TIMER2 underflow

T1 : TIMER1 underflow

DA : DAC timer underflow

1 : A bit value 1 indicates pending interrupt waiting to be serviced.

RESET: The IFR is left in the same state it was before RESET low, assuming no interruption in power.

† INT6 and INT7 may be associated instead with the Comparator function, if the Comparator Enable bit has been set. Refer to Section 3.3, *Comparator*, for details.

Individual interrupts are enabled or disabled for service by setting or clearing the respective bit in the interrupt mask register (IMR, 8 bits). If an interrupt level has its bit cleared in the IMR, then the interrupt service associated with that interrupt is disabled. Setting the bit in the IMR allows service to occur (pending the trigger-event which is registered in the IFR).

The IMR is accessible as part of another (larger) register, namely, the interrupt/general control register (peripheral port 0x38). After a RESET LOW, the default value of each bit in the IMR is zero: no interrupt service enabled. A full description of the bit locations in the interrupt/general control register can be found in Section 3.4, *Interrupt/General Control Register*.

The IMR functions independently of the IFR, in the sense that interrupt-trigger events can be registered in the IFR, even if the respective IMR bit is clear. Both the IFR and IMR are readable and writeable as port addressed registers. To read the register, use the IN instruction in conjunction with the port address (0x38 or 0x39). Use the OUT instruction to write. (Refer to Section 2.6.2, *Peripheral Communications (Ports)*, for more information.)

Note: Setting a Bit in the IFR Using the OUT Instruction

Setting a bit within the IFR using the OUT instruction is a valid way of obtaining a software interrupt. An IFR bit may also be cleared, using OUT, at any time.

Assuming the global interrupt enable is set and the specific bit within the IMR is set, then, at the time of the interrupt-trigger event, an interrupt service branch is initiated. (The trigger event is marked by a 0-to-1 transition in the IFR bit). At that time, the core processor searches all interrupt levels which have both: 1) pending interrupt flag, and 2) interrupt service enabled. The highest priority interrupt among these is selected. The program then branches to the location which is stored in the associated Interrupt Vector (Section 2.6.3, *Interrupt Vectors*). This location constitutes the start of the interrupt service routine. Instructions in the interrupt service routine are executed until the IRET (return) instruction is encountered. Afterwards, any other pending interrupts will be similarly serviced, in the order of their priority. Eventually, the program returns to whatever point it was before the first interrupt service branch.

When an interrupt service branch is taken, the global interrupt enable is automatically cleared by the core processor. This disables all further interrupt service branches while still in the pending service routine. As a result, the programmer must re-enable the interrupts globally using the INTE instruction. If performed as the second-to-last instruction in the service routine, then no nesting of multiple interrupts will occur. If, on the other hand, a nesting of certain interrupts is desired, then the INTE instruction may be included as the first instruction (or anywhere else) within the service routine.

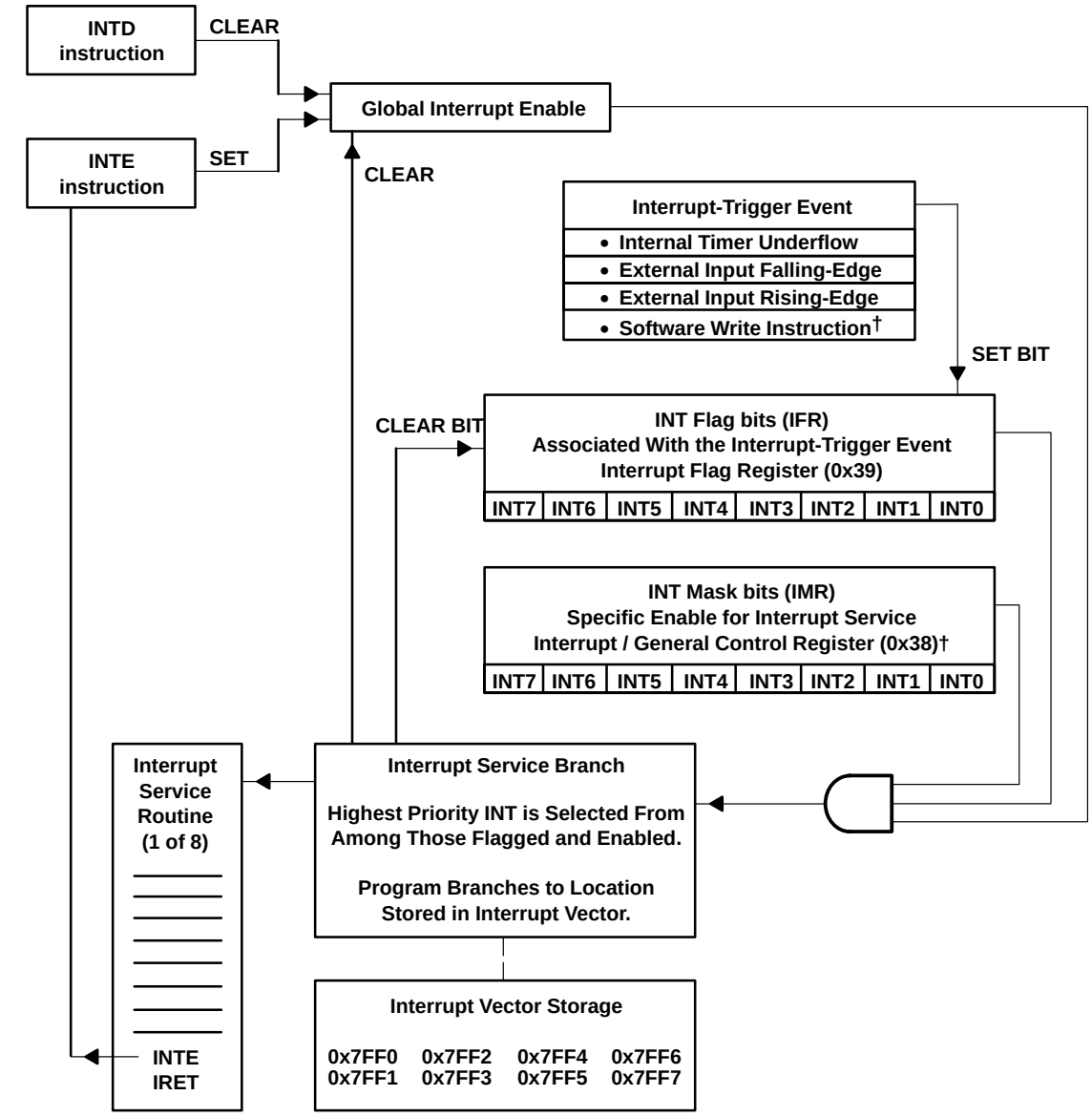
When an interrupt service branch is taken, the processor core also clears another status, namely, the respective bit in the IFR. This action automatically communicates to the IFR that the current pending interrupt is now being serviced. Once cleared, the IFR bit is ready to receive another SET whenever the next trigger event occurs for that interrupt.

Note: Interrupt Service Branch

If the interrupt service branch is **not** enabled by the respective bit in the mask register, then neither the global interrupt enable nor the respective flag bit is cleared. No program vectoring occurs.

Figure 2–8 provides an overview of the interrupt control sequence. INT0 is the highest priority interrupt, and INT7 is the lowest priority interrupt.

Figure 2–8. Interrupt Initialization Sequence



† The port-addressed write instruction (OUT) can be used to SET or CLEAR bits in the IFR and IMR.

In addition to being individually enabled, all interrupts must be GLOBALLY enabled before any one can be serviced. Whenever interrupts are globally disabled, the interrupt flag register may still receive updates on pending trigger events. Those trigger events, however, are not serviced until the next INTE instruction is encountered.

After an interrupt service branch, it is the responsibility of the programmer to re-SET the global interrupt enable, using the INTE instruction.

2.8 Clock Control

2.8.1 Oscillator Options

The C6xx has two oscillator options available. Either option may be enabled using the appropriate control bits in the clock speed control register (ClkSpdCtrl). The ClkSpdCtrl is described in Section 2.9.3, *Clock Speed Control Register*.

The first oscillator option, called the resistor-trimmed oscillator (RTO), is useful in low-cost applications where accuracy is less critical. This option utilizes a single external resistor to reference and stabilize the frequency of an internal oscillator. The oscillator is designed to run nominally at 32 kHz. It has a low V_{DD} coefficient and a low temperature coefficient (refer to the data sheet). The reference resistor is mounted externally across pins OSC_{IN} and OSC_{OUT}. The RTO oscillator is insensitive to variations in the lead capacitance at these pins. The required value of the reference resistor is 470 kΩ (1%).

The second oscillator option, CRO for crystal referenced, is a real time clock utilizing a 32.768 kHz crystal. The crystal is mounted externally across pins OSC_{IN} and OSC_{OUT}.

2.8.2 PLL Performance

A software controlled PLL multiplies the reference frequency (generated from either RTO or CRO) by integer multiples. This higher frequency drives the master clock which, in turn, drives the CPU clock. The master clock (MC) drives the circuitry in the periphery sections of the C6xx. The CPU Clock drives the core processor; its rate determines the overall processor speed. The multiplier in the PLL circuit, therefore, allows the master clock and the CPU clock to be adjusted between their minimum and maximum values.

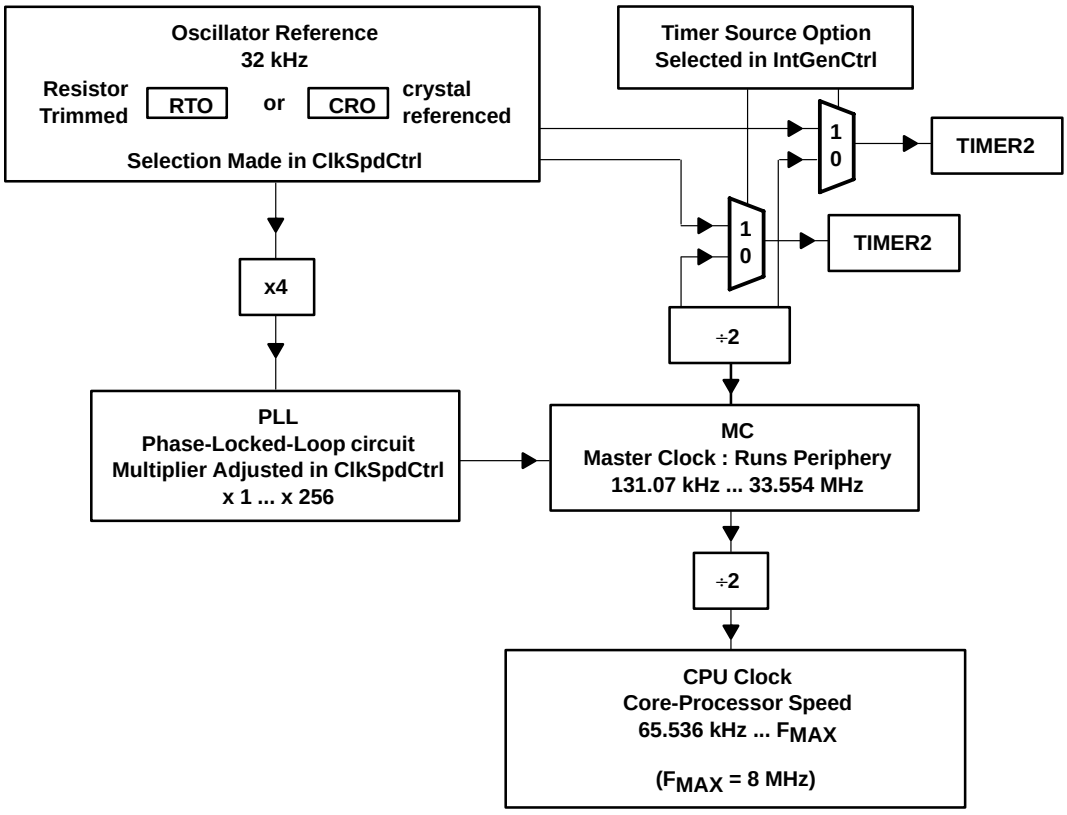
For either oscillator option, the reference frequency (32.768 kHz) is multiplied by four before it is accessed by the PLL circuit. The base frequency for the PLL,

therefore, is 131.07 kHz, and the multiplier operates in increments of this base frequency. The minimum multiplication of the base frequency is 1, and the maximum multiplication is 256. The resulting master clock frequency, therefore, can be varied from a minimum of 131.07 kHz to a maximum of 33.554 MHz, in 131.07 kHz steps.

From the master clock to the CPU clock, there is a divide-by-two in frequency. The CPU clock, therefore, can be set to run between 65.536 kHz and the maximum achievable (refer to the data sheet), in 65.536 kHz steps.

The maximum required CPU clock frequency for the C6xx is 8 MHz over the entire V_{DD} range. This rate applies to the speed of the core processor. Higher CPU clock frequencies may be achieved, but these are not qualified over the complete range of supply voltages in the guaranteed specification.

Figure 2–9. PLL Performance



2.8.3 Clock Speed Control Register

The ClkSpdCtrl is a 16-bit memory mapped register located at address 0x3D. The reference oscillator (RTO or CRO) is selected by setting one of the two control bits located at bits 8 and 9. Setting bit 8 configures the C6xx for the RTO reference option and simultaneously starts that oscillator. Setting bit 9 configures the C6xx for the CRO reference option and simultaneously pulses the crystal, which starts that oscillator.

Note: ClkSpdCtrl Bits 8 and 9

When bit 8 is set in the ClkSpdCtrl register, the crystal oscillator bit (bit 9) becomes the least significant bit of the 6-bit resistor trim value. Thus, bits 15–11 and 9 make up the 6-bit resistor trim value. For example, if the ClkSpdCtrl register is **00010**X**1**1XXXXXXXX (X means don't care, bold numbers are resistor trim bits), then the resistor trim value is equal to five.

The default value of the ClkSpdCtrl is 0x0000, which means that neither option is enabled by default. Immediately after a RESET LOW-to-HIGH, and regardless of whether a resistor or a crystal is installed across OSC_{IN}/OSC_{OUT}, the C6xx does **not** have a reference oscillator running. In the absence of a reference, however, the PLL still oscillates; it bottoms-out at a minimum frequency. The master clock, in turn, runs at a very slow frequency (less than 100 kHz) in the absence of a reference oscillator. Under this condition, program execution is supported at a slow rate until one of the two references (RTO or CRO) is enabled in software. (Refer to the data sheets for the MSP50Cxx devices).

Once a reference oscillator has been enabled, the speed of the master clock (MC) can be set and adjusted, as desired. Bits 7 through 0 in the ClkSpdCtrl constitute the PLL multiplier (PLLM). The value written to the PLLM controls the effective scaling of the MC, relative to the 131.07 kHz base frequency. A 0 value in PLLM yields the minimum multiplication of 1, and a 255 value in PLLM yields the maximum multiplication of 256. The resulting MC frequency, therefore, is controlled as follows:

MC Master clock frequency kHz = (PLLM register value + 1) $\times$ 131.07 kHz

CPU Clock frequency kHz = (PLLM register value + 1) $\times$ 65.536 kHz

The configuration of bits in the clock speed control register appears below:

ClkSpdCtrl register

address 0x3D (16-bit wide location)

WRITE only	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
	T5	T4	T3	T2	T1	I	C or T0	R	**M**	**M**	**M**	**M**	**M**	**M**	**M**	**M**

T : **RTO** oscillator-Trim adjust

I : Idle State clock Control

C : enable Crystal oscillator (or **T0** if **R** is set

R : enable Resistor-trimmed oscillator

M : **PLLM** multiplier bits for MC

0x0000 : default state after **RESET** LOW

Bit 10 in the ClkSpdCtrl is idle state clock control. The level of deep-sleep generated by the IDLE instruction is partially controlled by this bit. When this bit is cleared (default setting), the CPU clock is stopped during the sleep, but the MC remains running. When the idle state clock control bit is set, both the CPU clock and the MC are stopped during sleep. Refer to section 2.11 for more information regarding the C6xx's reduced-power modes.

Note: Reference Oscillator Stopped by Programmed Disable

If the reference oscillator is stopped by a programmed disable, then, on re-enable, the oscillator requires some time to restart and resume its correct frequency. This time imposes a delay on the core processor resuming full-speed operation. The time-delay required for the CRO to start is GREATER than the time-delay required for the RTO to start.

2.8.4 RTO Oscillator Trim Adjustment

Bits 15 through 11 and bit 9 (6 bits total) in the ClkSpdCtrl effect a software control for the RTO oscillator frequency. The purpose of this control is to trim the RTO to its rated (32 kHz) specification. The correct trim value varies from device to device. The user must program bits 15 through 11 and 9, in order to achieve the 32-kHz specification within the rated tolerances. Texas Instruments provides the trim value to the programmer of the P614 part with a sticker on the body of the chip. For the C6xx parts, the correct trim value is located at I/O location 0x2Fh.

RTRIM Register (Read Only) (Applies to MSP50C6xx Device Only)

I/O Address 0x2Fh (17-bit wide location)

16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
R	R	R	R	R	R	R	R	R	R	R	T5	T4	T3	T2	T1	T0

T: RTO oscillator-trim storage (device specific)

R: reserved for Texas Instruments use

ClkSpdCtrl Value Copied (Shaded)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
T5	T4	T3	T2	T1	I	T0	1	M7	M6	M5	M4	M3	M2	M1	M0

When selecting and enabling the RTO oscillator,therefore, the bits at positions 05 through 01 should be read from I/O location 0x2F (MSP50C6xx device only), then copied to the ClkSpdCtrl trim adjust (bits 15 through 11 of control register 0x3D), and bit 0 of 0x2F I/O port should be copied to bit 9 of ClkSpdCtrl register. The bit ordering is the same; bit 04 of I/O 0x2F copies to bit 15 of register 0x3D. Likewise, bit 00 of I/O 0x2F copies to bit 9 of register 0x3D.

However, the general specification of the adjustment can be useful in certain circumstances. For example, the adjustment can be used to obtain a programmatic increase or decrease in the speed of the RTO reference. The default value for the adjustment, after RESET low, is all zeros. The zero value generates the slowest programmable rate for the RTO reference. The maximum value, 0x3F, generates the fastest programmable rate for the RTO reference. The full range from 0x00 to 0x3F, effects an approximate +62% change (based on the RTO resistor value specification).

On the P614 part, the above method does not cause in the correct trim value to be loaded in ClkSpdCtrl. MSP50P614 is an EPROM device. Any preprogrammed value is erased when the chip goes through a UV erase procedure. The RTO trim value must, therefore, be computed separately for each chip. RTO trim values differ from one chip to another, is identical for the same chip.

Note: Register Trim Value

A resistor trim value is only needed when the resistor trimmed oscillator (RTO) is used. The MSP50P614 device must determine the trim value separately and use this value in the ClkSpdCtrl register bits 15–11 and 9, but C6xx device needs to copy bit 0 of I/O location 0x2F to bit 9 of the ClkSpdCtrl register and bits 5 through 1 to bits 15 through 11 of ClkSpdCtrl register.

This software-controlled trim for the RTO is **not** a replacement for the external reference-resistor mounted at pins OSC_{IN} and OSC_{OUT}. Also, note that this adjustment has no effect on the rate of the CRO reference oscillator.

2.9 Timer Registers

The C6xx contains two identical timers, TIMER1 and TIMER2. Each includes a period register and a count-down register. The period register (PRD1 or PRD2) defines the initial value for the counter, and the count-down register (TIM1 or TIM2) does the counting. When the count-down register decrements to the value 0x0000, then the value currently stored in the period register is loaded to the count-down register. The count-down register then resumes counting again from that value.

For each TIMER, there is an interrupt-trigger event associated with the TIMER's underflow condition (the point of reaching 0x0000 and then re-setting again). When enabled, the interrupt INT1 is triggered by the underflow of TIMER1, and the interrupt INT2 is triggered by the underflow of TIMER2. INT1 and INT2 are the second and third-highest priority interrupts in the C6xx. Refer to Section 2.7, *Interrupt Logic*, for a summary of the interrupt logic, and to Section 2.6.3, *Interrupt Vectors*, for a listing of the interrupt vectors.

Both the period and the count-down registers are readable and writeable as port-addressed registers:

(16-bit wide location)

	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
PRD1 register†	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P
address 0x3A	TIMER1 Period															
TIM1 register†	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
address 0x3B	TIMER1 Count-Down														Triggers INT1 on underflow	
PRD2 register	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P
address 0x3E	TIMER2 Period															
TIM2 register	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
address 0x3F	TIMER2 Count-Down														Triggers INT2 on underflow	

P : period register (initial counter value)

T : count-down register (counts from the value in P)

0x0000 : default state of both registers after RESET LOW

† TIMER1 may be associated with the comparator function, if the comparator enable bit is set. Refer to Section 3.3, *Comparator*, for details.

Reading from either the PRD or the TIM returns the current state of the register. This can be used to monitor the progress of the TIM register at any time.

Writing to the PRD register does not change the TIM register until the TIM register has finished decrementing to 0x0000. The new value in the PRD register is then loaded to the TIM register, and counting resumes from the new value.

Note: Writing to the TIM Register

Writing to the TIM register causes the same value to be written to the PRD register. In this case, the TIM register is immediately updated, and counting continues immediately from the new value.

Each TIMER decrements its count-down register at a fixed clock rate. The rate is selectable between two existing clock sources: the reference oscillator or 1/2 Master Clock. The rate of the master clock (MC) is programmable. It is determined by the value loaded to the PLL multiplier (Section 2.9.3, *Clock Speed Control Register*). The source to the TIMER is therefore one-half the frequency of the programmed master clock (1/2 MC). If, instead, the reference oscillator is selected as the source to the TIMER, then the source is either a resistor-trimmed oscillator (RTO) or a crystal oscillator (CRO). Both reference oscillators are designed to run at a nominal 32 kHz. Refer to Section 2.9, *Clock Control*, for more information regarding the oscillator configuration and clock programmability.

Selection between the timer-source options is made using two control bits in the interrupt/general control register (IntGenCtrl). The IntGenCtrl is a 16-bit port-addressed register at 0x38. Clearing bit 8 selects 1/2 MC as the source for TIMER1. Setting bit 8 selects the reference oscillator as the source for TIMER1. Similarly, clearing bit 9 of the IntGenCtrl selects 1/2 MC as the source for TIMER2. Setting bit 9 selects the reference oscillator as the source for TIMER2. The default value after a RESET LOW is zero: select 1/2 MC as the source.

Each of the TIMERs counts from the value stored in its period register to 0x0000. These maximum and minimum counts each receive a full clock cycle from the TIMER source. This means that the true period of the TIMER, from one underflow event to the next, is the value stored in the period register plus one:

Time duration btwn. underflows = (value in PRD + 1) ÷ (frequency of Timer Source)

TIMER1 and TIMER2 must be enabled for use. This is done at the IntGenCtrl register. Bit 10 of the IntGenCtrl is the enable bit for TIMER1, and bit 11 is the

enable bit for TIMER2. Setting the enable bit enables the TIMER, i.e., starts count-down running. Clearing the enable bit disables the TIMER, i.e., stops the count-down. The default setting after a RESET LOW is zero: both TIMERs disabled. Refer to Section 3.4, *Interrupt/General Control Register*, for summary information regarding the IntGenCtrl.

The TIMER enable bits may be used to start and stop the TIMERs repeatedly in software. Switching the enable bit from 1 to 0 stops the TIMER, but the current value in the count-down register is retained. When the enable bit is subsequently switched from 0 to 1, count-down then resumes from the held value. The following procedure outlines one (of many) possible ways to start the TIMERs. TIMER2 is given as an example:

1) Select the TIMER2 clock source: 1/2 MC or RTO/CRO (bit 9 of the IntGenCtrl, address 0x38).

2) Clear the TIMER2 enable (bit 11 in the IntGenCtrl).

3) Load the count-down register (TIM2) with the desired period value ahead-of-time. This prepares TIM2 for counting, and also loads the period register (PRD2) with its value.

4) Be sure the TIMER2 interrupt (INT2) has been enabled for service (set bit 2 of IntGenCtrl).

5) Flip the TIMER2 enable bit from 0 to 1, at the precise time you want counting to begin.

2.10 Reduced Power Modes

The power consumption of the C6xx is greatest when the DAC circuitry is called into operation, i.e., when the synthesizer speaks. There are, however, a number of reduced power modes (sleep states) on the C6xx which may be engaged during quiet intervals.

The performance and flexibility of the reduced power modes make the C6xx ideal for battery powered operation. Refer to data sheets for the MSP50C6xx devices.

The reduced power state on the C6xx is achieved by a call to the IDLE instruction. The idle state is released by some interrupt event. Different modes (or levels) of reduced-power are brought about by controlling a number of different core and periphery components on the device. These components are independently enabled/disabled **before** engaging the IDLE instruction. The number of subsystems left running during sleep directly impacts the

overall power consumption during that state. The various subsystems that determine (or are affected by) the depth of sleep include the:

- Processor core, which is driven by the CPU clock
- PLL clock circuitry
- PLL reference oscillator
- C6xx periphery, which is driven by the master clock
- TIMER1 and TIMER2
- PDM pulsing

The deepest sleep achievable on the C6xx, for example, is a mode where all of the previously listed subsytems are stopped. In this state, the device draws less than 10 μA of current and obtains the greatest power savings. It may be awakened from this state using an external interrupt (input port).

A number of control parameters determine which of the internal components are left running after the IDLE instruction. In most cases, the states of these controls may be mixed in any combination. There are three combinations, however, which are primarily useful. The three modes (light, mid, and deep sleep) are executed through the independent control of two bits: 1) the idle state clock control, and 2) the reference oscillator enable. The other pertinent controls simply enhance the performance of the modes dictated by these two. Table 2–3 gives a listing of all of the controls which should be maintained by the programmer before engaging the IDLE instruction. In some cases, it will be impossible to wake from sleep unless certain controls are set appropriately before going to sleep. (In those cases, only the hardware RESET low-to-high will bring the device back into its normal operating state.)

The top row in Table 2–3 lists the first of the two primary controls, namely, the idle state clock control. The idle state clock control determines the status of the master clock (MC) during sleep. Setting the idle state control causes the CPU clock, the PLL clock circuitry, and the MC to stop after the next IDLE instruction. Clearing the idle state control causes only the CPU clock to stop after IDLE. The PLL clock circuitry governs the MC and determines its rate. Whenever the PLL circuitry is suspended, therefore, the MC stops. The idle state clock control is accessed at bit 10 in the ClkSpdCtrl register (refer to Section 2.8.3, *Clock Speed Control Register*, for more information).

The reference oscillator enable is the other control which selects between the three reduced power modes listed in Table 2–3. This control may be one of two bits, depending on which oscillator reference is implemented in circuitry (refer to Section 2.8.3, *Clock Speed Control Register*). When using the resistor-trimmed oscillator (RTO), the reference oscillator enable appears as bit 8 in the ClkSpdCtrl register. When using the crystal-referenced oscillator (CRO), the reference oscillator enable appears as bit 9 in the ClkSpdCtrl register. If both bits 8 and 9 are clear, then no reference oscillator is enabled.

If either of bits 8 or 9 are set, then the reference oscillator enable is considered set. This enables the PLL circuitry to regulate to the reference frequency, 32 kHz (assuming the idle state clock control is clear). Whichever state the reference oscillator is in before idle, it remains in that state (running or stopped) after idle. If the reference oscillator is left running during sleep, however, it comes at a cost to power consumption. (This may be a necessary cost if, in your application, elapsed time needs to be monitored during sleep.)

The power consumed during sleep when the RTO oscillator is left running is *greater* than the power consumed during sleep when the CRO oscillator is left running.

If the idle state clock control is clear, then the PLL circuitry, active during sleep, will attempt to regulate the MC to whatever frequency is programmed in the PLL multiplier (see Section 2.9.3, *Clock Speed Control Register*). The MC continues to run at this frequency, even during sleep, provided that the reference oscillator is enabled.

If the idle state clock control is set, then neither the MC, CPU clock, nor the TIMER clocks run during sleep, unless the TIMER source is linked to the reference oscillator (Section 2.8, Time Registers). These relationships are shown explicitly, as a function of the reduced power mode, in Table 2–4.

Because the DAC circuitry is the single most source of power consumed on the C6xx, it is important to disable the DAC entirely before engaging any IDLE instruction. This is accomplished at the DAC control register, address 0x34. Refer to Section 3.2.2, *DAC Control and Data Registers*.

The ARM bit is another important control to consider before engaging the reduced power mode. It is recommended that the ARM bit be cleared whenever the idle state clock control is clear, and set whenever the idle state clock control is set. The set ARM bit causes an asynchronous response to all programmable interrupts when in the sleep state. (The cleared ARM bit yields the standard synchronous response at all times.) Affected interrupts include those tied to TIMER1 and TIMER2, as well as those tied to the inputs at Ports F, D_2, D_3, D_4, and D_5. The advantage to having the ARM bit set is that the device may be awakened by one of these interrupts, even when the PLL clock circuitry is stopped in sleep (by virtue of the idle state control). The disadvantage of the asynchronous response, however, is that it can render irregularities in the timing of response to these same inputs.

Note: Idle State Clock Control Bit

If the idle state clock control bit is set and the ARM bit is clear, the **only** event that can wake the C6xx after an IDLE instruction is a hardware RESET low-to-high. When at sleep, the device will **not** respond to the input ports, nor to the internal timers.

Table 2–3. Programmable Bits Needed to Control Reduced Power Modes

		⟶ **deeper sleep** … **relatively less power** ⟶		
Control Bit	**Label for Control Bit**	**LIGHT**	**MID**	**DEEP**
Idle state clock control bit 10 **ClkSpdCtrl** register (0x3D)	A	0	1	1
Enable reference oscillator bit 09 : **CRO** or bit 08 : **RTO** **ClkSpdCtrl** register (0x3D)	B	1	1	0
ARM bit 14 **IntGenCtrl** register (0x38)	C	0	1	1
Enable PDM pulsing bit 02 **DAC Control** register (0x34)	D	Should be cleared before **any** IDLE instruction.		
IDLE instruction (executes the mode)	E	Same instruction is used to engage any of the modes.		
PLL multiplier bits 07 through 00 **ClkSpdCtrl** register (0x3D)	F	Programmed value is 0 … 255 .		

Table 2–4. Status of Circuitry When in Reduced Power Modes (Refer to Table 2–3)

		⟶ **deeper sleep** … **relatively less power** ⟶		
Component	**Determined by Controls**	**LIGHT**	**MID**	**DEEP**
CPU clock (processor core)	E	stopped	stopped	stopped
PLL clock circuitry	A, E	running	stopped	stopped
Master clock (MC) status (C6xx periphery)	A, E	running	stopped	stopped
MC rate	B, F	131 kHz … 34 MHz	—	—
Synchrony of external interrupts	C, E	Synchronous	Asynchronous	Asynchronous
PDM pulsing	D	stopped	stopped	stopped
TIMER1 or TIMER2 status • Assuming TIMER is enabled 1) TIMER source = 1/2 MC 2) TIMER source = RTO or CRO	A, B, E	1) running 2) running	1) stopped 2) running	1) stopped 2) stopped

If the reference oscillator is stopped by a programmed disable or by an IDLE instruction, then, on re-enable or wake-up, the oscillator requires some time to restart and resume its correct frequency. This time imposes a delay on the core processor resuming full-speed operation. The time-delay required for the CRO to start is *greater* than the time-delay required for the RTO to start.

There are a number of ways to wake the C6xx from the IDLE-induced sleep state. The various options are summarized, as a function of the reduced power mode, in Table 2–5. Naturally, the RESET event (happens after the RESET pin has gone low-to-high) causes an immediate escape from sleep; whereby, the program counter assumes the location stored in the RESET interrupt vector. The RESET escape from sleep is always enabled, regardless of the depth of sleep or the state of programmable controls.

The more functional methods available for waking the device are: 1) the Internal TIMER interrupt, and 2) the external input-port interrupt. For either of these options to work, the respective bit in the interrupt mask register (address 0x38) must be set to enable the associated interrupt service. If the appropriate IMR bit is not set before the IDLE instruction, then the interrupt-trigger event will not be capable of waking the device from sleep. Note also the state of the idle state clock control bit and the ARM bit, if you expect to wake-up using

either type of interrupt (internal or external). In most cases, the state of these bits should coincide.

The interrupt-trigger event associated with each of the two internal TIMERs is the underflow condition of the TIMER. In order for a TIMER underflow to occur during sleep, the TIMER must be left running before going to sleep. In certain cases, however, the act of going to sleep can bring a TIMER to stop, thereby preventing a TIMER-induced wake-up. The bottom row of Table 2–4 illustrates the various conditions under which the TIMER will continue to run after the IDLE instruction. Note that the reduced power mode DEEP leaves both TIMERs stopped after IDLE. This mode cannot, therefore, be used for a timed wake-up sequence.

Table 2–5. How to Wake Up from Reduced Power Modes (Refer to Table 2–3 and Table 2–4)

		⟶ **deeper sleep** … **relatively less power** ⟶		
Event	**Determined by Controls**	**LIGHT**	**MID**	**DEEP**
Timer interrupts TIMER1 and TIMER2 • Assuming respective IMR bit is set • Assuming ARM bit is set as in *C*	A, B, C	If TIMER is running, then Underflow wakes device.		No wake-up from TIMER.
External interrupts Port F and $D_{2,3,4,5}$ (if input) • Assuming respective IMR bit is set • Assuming ARM bit is set as in *C*	C	Rising-Edge, or Falling-Edge, as appropriate, wakes device.		
RESET	none	RESET LOW-to-HIGH always wakes device.		
DAC Timer • Assuming PDM bit is clear as in *D*	D	No wake-up from DAC Timer.		

The external interrupt is the other programmable option for waking the C6xx from sleep. The associated interrupt-trigger event is, in some cases, a rising-edge at the input port; in some cases it is a falling-edge. Refer to Section 3.1.5, *Internal and External Interrupts*, for a full description of these events. Consider also the comparator driven interrupts described in Section 3.3, *Comparator*. The input ports which are supported by external interrupt include the entire F Port, and, when programmed as inputs, Ports D_2, D_3, D_4, and D_5. Refer to Section 3.1, *I/O*, for a description of the various I/O configurations.

Under normal operation the DAC timer, when IMR enabled, triggers an interrupt on underflow. Before any IDLE instruction, however, the entire DAC circuitry should be disabled. This ensures the effectiveness of the reduced power mode and prevents any wake-up from the DAC timer.

In order to wake the device using a programmable interrupt, the interrupt mask register must have the respective bit set to enable interrupt service (see Section 2.7, *Interrupt Logic*). In some cases, the ARM bit must also be set, in order for the interrupts to be visible during sleep.

After the C6xx wakes from sleep, the program counter assumes a specific location, resuming normal operation of the device. Normally, the destination of the program on wake-up is the interrupt service routine associated with the interrupt which initiated the wake-up. The start of the interrupt service routine is defined by the program location stored in the respective interrupt vector (see Section 2.6.3, *Interrupt Vectors*). This wake-up response requires that the global interrupt enable is set before going to sleep (use the INTE instruction).

If the global interrupt enable is CLEAR before going to sleep, then the programmed interrupt can still wake the device, provided that the respective IMR and ARM bits are set as in Table 2–3. The program counter returns to the location immediately following the IDLE instruction. This wake-up response may be useful for putting the C6xx into a hold sleep, where any number of programmable interrupts can wake the device. To accomplish this, the appropriate interrupts should be enabled in the IMR. Table 2–6 lists the possible destinations of the program counter on wake-up.

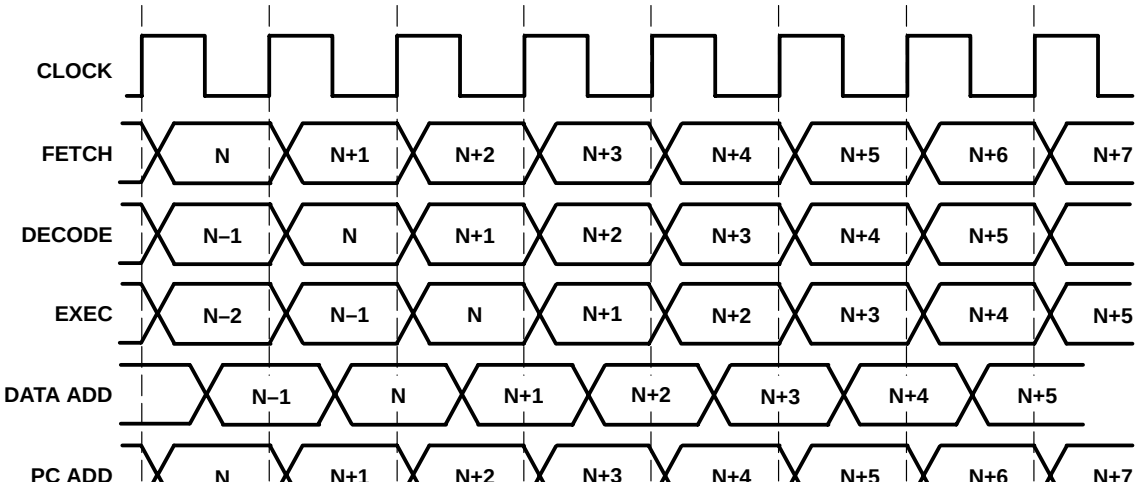
Table 2–6. Destination of Program Counter on Wake-Up Under Various Conditions

State of Interrupt Controls before IDLE Instruction	**Assuming Wake-Up can occur… Destination of Program Counter after Wake-Up**
• Global interrupt enable is SET • Respective IMR bit is SET	Program counter goes to the location stored in the interrupt vector associated with the waking Interrupt.
• Global interrupt enable is CLEAR • Respective IMR bit is SET	Program counter goes to the next instruction immediately following the IDLE which initiated sleep.
• Global interrupt enable is SET • Respective IMR bit is CLEAR	Wake-up cannot occur from the programmed Interrupt under these conditions.
	If RESET low-to-high occurs, then program goes to the location stored in the RESET interrupt vector.

2.11 Execution Timing

For executing program code, the C6xx's core processor has a three-level pipeline. The pipeline consists of instruction fetch, instruction decode, and instruction execution. A single instruction cycle is limited to one program Fetch plus one data memory read or write. The master clock consists of two phases with non-overlap protection. A fully static implementation eliminates pre-charge time on busses or in memory blocks. This design also results in a very low power dissipation. Figure 2–10 illustrates the basic timing relationship between the master clock and the execution pipeline.

Figure 2–10. Instruction Execution and Timing



Chapter 3

Peripheral Functions

This chapter describes in detail the MSP50C6xx peripheral functions, i.e., I/O control ports, general purpose I/O ports, interrupt control registers, comparator and digital-to-analog (DAC) control mechanisms.

Topic **Page**

3.1	**I/O**	**3–2**
3.2	**Digital-to-Analog Converter (DAC)**	**3–9**
3.3	**Comparator**	**3–15**
3.4	**Interrupt/General Control Register**	**3–18**
3.5	**Hardware Initialization States**	**3–20**

3.1 I/O

This section discusses the I/O capabilities of the MSP50C6xx family. The following table shows the number and types of I/O available on each device. Please note that this section discusses all I/O ports, which are only available on the MSP50C614 device. All other devices have only a subset of the I/O that is available on the MSP50C614.

Device	Ports Available	No. of General Purpose I/O	No. of Dedicated Inputs	No. of Dedicated Outputs
MSP50C614	A,B,C,D,E,F,G	40	8	16
MSP50C604	C,D	16	0	0
MSP50C605	C,D,E,F	24	8	0
MSP50C601	C,D,E,F	24	8	0

3.1.1 General-Purpose I/O Ports

The forty configurable input/output pins are organized in 5 ports, A,B,C,D, and E. Each port is one byte wide. The pins within these ports can be individually programmed as input or output, in any combination. The selection is made by clearing or setting the appropriate bit in the associated control register (Control A, B, C, D, or E). Clearing the bit in the control register renders the pin as a high-impedance input. Setting the control bit renders the pin as a totem-pole-output.

When configured as an input, the data presented to the input pin can be read by referring to the appropriate bit in the associated data register (Data A, B, C, D, or E). This is done using the IN instruction, with the address of the data register as an argument.

When configured as an output, the data driven by the output pin can be controlled by setting or clearing the appropriate bit in the associated data register. This is done using the OUT instruction, with the address of the data register as an argument.

	Port A	Port B	Port C	Port D	Port E
Control register address	0x04h†	0x0Ch	0x14h	0x1Ch	0x24h
Possible control values	0 = High-Z INPUT 1 = TOTEM-POLE OUTPUT				
Value after RESET low	0 = High-Z INPUT				
Data register address	0x00h	0x08h	0x10h	0x18h	0x20h
Possible input data values	Low = 0 High = 1 (don't care on write)				
Possible output data values	0 = Low 1 = High				

† Each of these I/O ports is only 8 bits wide. The reason for the 4-byte address spacing is so that instructions with limited addressability (such as memory transfers) can still access these registers.

Note: Reading the Data Register

Whether configured as input or as output, reading the data register reads the actual state of the pin.

The state of the control registers is initialized to 0x00 when the RESET pin is taken low. This puts all of the programmable I/O pins into an input state. This condition is maintained after RESET is taken high, and until the control registers are modified. The state of the data registers is not initialized with RESET. After RESET is taken high, the state of the data registers is unknown and must be initialized using software.

The 8-bit width is the true size of the mapped location. This is independent of the address spacing, which is greater than 8-bits. When writing to any of the locations in the I/O address map, therefore, the bit-masking need only extend across 8 bits. Within a 16-bit accumulator, the desired bits should be right-justified. When reading from these locations to a 16-bit accumulator, the IN instruction automatically clears the extra bits in excess of 8. The desired bits in the result will be right-justified within the accumulator.

The following table shows the bit locations of the I/O port mapping:

	(8-bit wide location) 07 06 05 04 03 02 01 00
A port data register ..... address 0x00	A7 A6 A5 A4 A3 A2 A1 A0
A port control register ... address 0x04	C C C C C C C C
B port data register ..... address 0x08	B7 B6 B5 B4 B3 B2 B1 B0
B port control register ... address 0x0C	C C C C C C C C
C port data register ..... address 0x10	C7 C6 C5 C4 C3 C2 C1 C0
C port control register ... address 0x14	C C C C C C C C
D port data register ..... address 0x18	D7 D6 D5 D4 D3 D2 D1 D0
D port control register† .. address 0x1C	C C C C C C C C
E port data register ..... address 0x20	E7 E6 E5 E4 E3 E2 E1 E0
E port control register ... address 0x24	C C C C C C C C
A7, B7, C7, D7, E7 : data register C : control register (0 = IN, 1 = OUT) 0x00 : state of control register after RESET low	

† Ports D_4 and D_5 may be dedicated to the Comparator function, if the Comparator Enable bit is set. If so, then bits 4 and 5 of the D port Control register *must* be CLEAR. Please refer to Section 3.3, *Comparator*, for details.

Port D_0 is connected to the branch condition COND1. Port D_1 is connected to the branch condition COND2, assuming the comparator is disabled. Please refer to Section 3.1.4, *Branch on D Port*, (and to Section 3.3, *Comparator*) for more information. External interrupts can be detected when transitions occur on ports D_2, D_3, D_4, and D_5. The interrupts associated with the D port are supported whether those pins are programmed as inputs or as outputs.

3.1.2 Dedicated Input Port F

Port F is an 8-bit wide input-only port. The data presented to the input pin can be read by referring to the appropriate bit in the F port data register, address 0x28. This is done using the IN instruction, with the 0x28 address as an argument. The state of the F port data registers is not initialized with RESET. After RESET is taken high, the state of the F port data register is unknown.

Each of the pins at port F has a programmable pull-up resistor. All eight pullup resistors can be enabled by setting the enable pullup (EP) in the interrupt/general control register (IntGenCtrl). The address of the IntGenCtrl is 0x38, and the location of the EP bit is 12. Clearing the EP bit disables the eight pullups,

and setting the EP bit enables the eight pullups. After RESET low, the default setting for the EP bit is 0 (F-port pullups disabled).

	Input Port F
Data register address	0x28h
Possible input data values	Low = 0 High = 1
Possible output data values	N/A
Value after RESET low	Pullup resistors DISABLED

When reading from the 8-bit F-port data register to a 16-bit accumulator, the IN instruction automatically clears the extra bits in excess of 8. The desired bits in the result will be right-justified within the accumulator.

The following table shows the bit locations of the port **F** address mapping:

F port Input Data register address 0x28h READ only	(8-bit wide location) 07 06 05 04 03 02 01 00 F7 F6 F5 F4 F3 F2 F1 F0

The external interrupt INT5 is triggered by a falling-edge event on any of the eight port-F input pins (see Section 3.1.5, *Internal and External Interrupts*). The F port input pins are gated through an eight-input AND gate, such that any input pin going low causes the output of the AND gate to go low. Therefore, if any input pin is held low, the device will not trigger INT5 when another input is taken low. Specifically, INT5 is triggered if all eight port-F pins are held high, and then one or more of these pins is taken low. This allows port F to be especially useful as a key-scan interface.

3.1.3 Dedicated Output Port G

Port G is a 16-bit wide output-only port. The output drivers have a Totem-Pole configuration. The data driven by the output pin can be controlled by setting or clearing the appropriate bit in the G port data register, address 0x2C. This is done using the OUT instruction, with the 0x2C address as an argument. The port G outputs are set to 0 (logic low) when the RESET pin is taken low. This condition is maintained after RESET is taken high, and until the G port data register is modified.

	Totem-Pole Output Port G
Data register address	0x2Ch
Possible input data values	N/A
Possible output data values	0 = Low 1 = High
Value after RESET low	0 = Low

The following table shows the bit locations of the port **G** address mapping:

G port Data address 0x2C read and write	(16-bit wide location) 15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
	G15	G14	G13	G12	G11	G10	G9	G8	G7	G6	G5	G4	G3	G2	G1	G0

0x0000 : default state of data register after RESET low

3.1.4 Branch on D Port

Instructions exist to branch conditionally depending upon the state of ports D_0 and D_1. These conditionals are COND1 and COND2, respectively. The conditionals are supported whether the D_0 and D_1 ports are configured as inputs or as outputs. The following table lists the four possible logical states for D_0 and D_1, along with the software instructions affected by them.

	$D_0 = 1$	COND1 = TRUE. . .	CIN1	has its conditional call taken.
			CNIN1	has its conditional call ignored.
			JIN1	has its conditional jump taken.
			JNIN1	has its conditional jump ignored.
	$D_0 = 0$	COND1 = FALSE. . .	CIN1	has its conditional call ignored.
			CNIN1	has its conditional call taken.
			JIN1	has its conditional jump ignored.
			JNIN1	has its conditional jump taken.
†	$D_1 = 1$	COND2 = TRUE. . .	CIN2	has its conditional call taken.
			CNIN2	has its conditional call ignored.
			JIN2	has its conditional jump taken.
			JNIN2	has its conditional jump ignored.
†	$D_1 = 0$	COND2 = FALSE. . .	CIN2	has its conditional call ignored.
			CNIN2	has its conditional call taken.
			JIN2	has its conditional jump ignored.
			JNIN2	has its conditional jump taken.

† COND2 may be associated instead with the comparator function, if the comparator Enable bit is set. Please refer to Section 3.3, *Comparator*, for details.

3.1.5 Internal and External Interrupts

INT3, INT4, INT6, and INT7 are external interrupts which may be triggered by events on the PD_2, PD_3, PD_4, and PD_5 pins. These interrupts are supported whether the D-port pins are programmed as inputs or outputs. (When programmed as an output, the pin effectively triggers a software interrupt.)

INT5 is an external interrupt triggered by a falling-edge event on any of the F-port inputs. It is triggered if all eight port-F pins are held high, and then one or more of these pins is taken low.

Only the transition from 0xFFh (all high) to (one or more pins) low will trigger the INT5 event. If any F-port pin is continuously held low and another is toggled high-to-low, no interrupt is detected at the toggling pin. After all F-port pins have been brought high again, then it is possible for a new INT5 trigger to occur.

INT0 is an internal interrupt (highest priority) which is triggered by an underflow condition on the DAC Timer (see Section 3.2.2, *DAC Control and Data Registers*). INT1 and INT2 are high-priority, internal interrupts triggered by the underflow conditions on TIMER1 and TIMER2, respectively. Please refer to Section 2.8, *Timer Registers*, for a full description of the TIMER controls and their underflow conditions.

When properly enabled, any of these interrupts may be used to wake the device up from a reduced-power state. In a deep-sleep state, they can also be used to wake the device when used in conjunction with the ARM bit. Please refer to Section 2.11, *Reduced Power Modes*, for information regarding the MSP50C6xx's reduced power modes.

A summary of the interrupts is given in Table 3–1.

Table 3–1. Interrupts

Interrupt	Vector	Source	Trigger Event	Priority	Comment
INT0	0x7FF0	DAC Timer	Timer underflow	Highest	Used to synch. speech data
INT1	0x7FF1	TIMER1	Timer underflow	2nd	
INT2	0x7FF2	TIMER2	Timer underflow	3rd	
INT3	0x7FF3	PD_2	Rising edge	4th	Port D_2 goes high
INT4	0x7FF4	PD_3	Falling edge	5th	Port D_3 goes low
INT5†	0x7FF5	All port F	Any falling edge	6th	Any F port pin goes from all-high to low
INT6‡	0x7FF6	PD_4	Rising edge	7th	Port D_4 goes high
INT7‡	0x7FF7	PD_5	Falling edge	Lowest	Port D_5 goes low

† All F port pins must be high previous to one or more going low.
‡ INT6 and INT7 may be associated with the Comparator function, if the Comparator Enable bit has been set.

Note: Interrupts in Reduced Power Mode

An interrupt may be lost if its event occurs during power-up or wake-up from a reduced power mode. Also, note that interrupts are generated as a divided signal from the master clock. The frequency of the various timer interrupts will therefore vary, depending upon the operating master clock frequency.

3.2 Digital-to-Analog Converter (DAC)

The MSP50C6xx incorporates a two-pin pulse-density-modulated DAC which is capable of driving a 32-Ω loudspeaker directly. To drive loud speakers other than 32 Ω, an external impedance-matching circuit is required.

3.2.1 Pulse-Density Modulation Rate

The rate of the master clock (MC) determines the pulse-density-modulation (PDM) rate, and this governs the output sampling-rate and the achievable DAC resolution. In particular, the sampling rate is determined by dividing the PDM rate by the required resolution:

$$\text{Output sampling rate} = \text{PDM Rate} \div 2^{(\#\text{ DAC resolution bits})}$$

PDM Rate	#DAC resolution bits
Set in ClkSpdCtrl register	Set in DAC control register
Address 0x3D	Address 0x34

For example, a 9 bit PDM DAC at 8 kHz sampling rate requires a PDM rate of 4.096 MHz.

There are four sampling rates which may be used effectively within the constraints of the MSP50C6xx and the various software vocoders provided by Texas Instruments. These are: 7.2 kHz, 8 kHz, 10 kHz, and 11.025 kHz. Other sampling rates, however, may also be possible.

From the MC to the PDM clock, there is an *optional* divide-by-two in frequency. This option is controlled by the PDM clock divider in the interrupt/general control register. This means that the PDM rate can be set to run between 131.07 kHz and 33.554 MHz in 131.07 kHz steps (the same as the MC). Or, the PDM rate can be set to run between 65.536 kHz and the maximum achievable CPU frequency (see the MSP50C6xx data sheet (SPSS023)*, Electrical Specifications*) in 65.536-kHz steps. The PDM clock divider determines which of these two ranges apply. Within these ranges, it is the PLLM that sets the rate: ClkSpdCtrl, 0x3D. Refer to Section 3.2.3, *PDM Clock Divider*, for more information regarding the PDM clock divider and the available combinations of CPU clock rates vs sampling rates. (Section 2.9.3, *Clock Speed Control Register*, contains more details regarding the PLLM.)

3.2.2 DAC Control and Data Registers

The resolution of the PDM-DAC is selected using the control bits in the DAC control register (address 0x34). The available options are 8, 9, or 10 bits of resolution. Bits 0 and 1 in the DAC control register control this option:

DAC Control register Address 0x34	(4-bit wide location) 03 02 01 00
Set DAC resolution to 8 bits: Set DAC resolution to 9 bits: Set DAC resolution to 10 bits:	DM E 0 0 DM E 0 1 DM E 1 0
DM : Drive Mode selection (0 = C3x style : 1 = C5x style) E : pulse-density-modulation Enable (overall DAC enable) 0x0 : default state of register after RESET low	

Bit 2 in the DAC control register is used to enable/disable the pulse-density modulation. This bit must be set in order to enable the overall functionality of the DAC. After RESET is held low, the default state of bit 2 is clear. In this state, the output at the DAC pins is guaranteed to be zero (no PDM pulsing). During DAC activity, the PDM enable bit may also be toggled at any time to achieve the zero state. In other words, toggling the PDM enable bit from high-to-low-to-high brings the DAC output to the known state of zero.

Note: PDM Enable Bit

By default, the PDM enable bit is cleared: DAC function is off.

Data values are output to the DAC by writing to the DAC data register, address 0x30. The highest-priority interrupt, INT0, is generated at the sampling rate governed by the ClkSpdCtrl and the DAC control register. The program in software is responsible for writing a correctly-scaled DAC value to the DAC data register, in response to each INT0 interrupt. The register at 0x30 is 16-bits wide. The data is written in sign-magnitude format. Bit 15 of the register is the sign bit. Bits 14 and 13 are the overflow bits. Bits 12 through 3 are the data-value bits: The MSB is bit 12, and the LSB is bit 5, 4, or 3, depending on the resolution.

DAC Data register

Address 0x30 (16-bit wide location)

Write Only	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
10 bit DAC resolution:	S	O	O	M	D	D	D	D	D	D	D	D	L	X	X	X
9 bit DAC resolution:	S	O	O	M	D	D	D	D	D	D	D	L	X	X	X	X
8 bit DAC resolution:	S	O	O	M	D	D	D	D	D	D	L	X	X	X	X	X

S : Sign bit M : Most-significant data value D ; Data (magnitude)
O : Overflow bits L : Least-significant data value X : ignored bits

The overflow bits function in different ways, depending on the drive mode selected. The two DAC drive modes are informally named *C3x style* and *C5x*

style. Their selection is made at bit 3 of the DAC control register (0x34). The *C3x style* is selected by clearing bit 3, and the *C5x style* is selected by setting bit 3. The default value of the selection is zero which yields the *C3x style*.

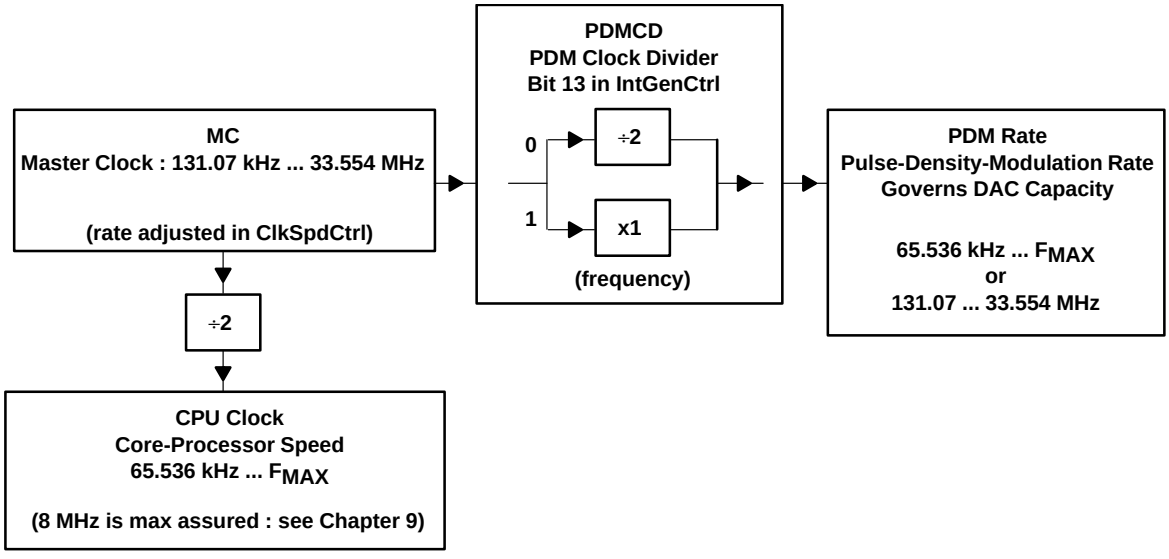
The overflow bits appear in the DAC data register (14 and 13) to the left of the MSB data bit (12). In the *C3x style* mode, the overflow bits serve as a 2-bit buffer to handle overflow in the value field (bits 12…3). Any magnitude written to the value field which is greater than 1023 (up to the limit 4095) lands a 1 in the overflow. The overflow state (when a 1 appears in either bit 13 or 14) yields the maximum PDM saturation and delivers the maximum possible current drive to the loudspeaker. The overflow bits thus help to ensure that the audible artifacts of *wrap-around* do not occur.

3.2.3 PDM Clock Divider

The pulse-density-modulation rate is determined by the master clock. The PDM rate may be set equal to the rate of the MC, or it may be set at one-half the rate of the MC. This option is controlled by the PDM clock divider (PDMCD) in the interrupt/general control register (IntGenCtrl). The PDMCD is located at bit 13 in IntGenCtrl (address 0x38).

Clearing the PDMCD bit results in a PDM rate equal to 1/2 MC (i.e., the CPU Clock rate). Setting the PDMCD bit results in a PDM rate equal to the MC. After RESET is held low, the default setting for the PDMCD bit is zero (PDM rate = 1/2 MC).

Figure 3–1. PDM Clock Divider



For a given sampling rate and DAC resolution, the CPU clock rate may be increased, if necessary, through the use of over-sampling. In the previous example, an original sampling rate of 8 kHz and a PDM rate of 4 MHz was used. A 2-times over-sampling, therefore, would require the PDM rate to be 8 MHz. This can be accomplished in two ways:

PDM rate = 8 MHz : Set the master clock to 8 MHz also (ClkSpdCtrl).
Set the PDMCD bit to 1: 1x master clock (IntGenCtrl).
CPU clock rate will be 4 MHz.

PDM rate = 8 MHz : Set the master clock to 16 MHz.
Set the PDMCD bit to 0: 1/2 master clock.
CPU clock rate will be 8 MHz.

In the case of over-sampling, the same number of instructions are achievable between each INT0 interrupt. Not every INT0, however, requires an independently computed synthesis value, hence, the advantage in increased instruction capacity. A 2-times over-sampling means that every 2nd INT0 requires a computed update from the synthesis algorithm. The other INT0 may be satisfied with an interpolating filter computation, then a return to the main program.

As stated previously, the maximum ensured CPU clock frequency for the MSP50C6xx operates over the entire V_{DD} range. This rate applies to the speed of the core processor. Operating the processor higher than the listed specification is not recommended by Texas Instruments.

The following tables illustrate a number of possible combinations with respect to sampling rate, PDM rate, DAC resolution, master clock rate, and CPU clock rate. The first table applies to the 8 kHz sampling rate and N-times-8 kHz over-sampling. The second applies to the 10 kHz sampling rate and N-times-10 kHz over-sampling.

Note:

The value programmed to the PLLM register is not exactly the multiplicative factor between the 32-kHz reference and the master clock. Refer to Section 2.9.3, *Clock Speed Control Register*, for more information on the relationship between the PLLM and the resulting MC rate.

The column in these tables output sampling rate reports the true audio sampling rate achievable by the MSP50C6xx, using the 32.768-kHz CRO. The values reported are not always exact multiples of the 8-kHz and 10-kHz options; however, they are the closest obtainable (using the PLLM multiplier) under the given set of constraints.

Example 3–1. 8-kHz Sampling Rate

8 kHz Nominal Synthesis Rate									
32.768 kHz Oscillator Reference									
DAC Precision	**IntGenCtrl PDMCD Bit**	**Over-Sampling Factor**	**ClkSpdCtrl PLLM Register Value (hex)**	**Master Clock Rate (MHz)**	**PDM Rate (MHz)**	**CPU Clock Rate (MHz)**	**Output Sampling Rate (kHz)**	**Number of Instructs Between DAC Interrupts**	**Number of Instructs Between 8 kHz Interrupts**
8 bits	1	1x	0x 0F	2.10	2.10	1.05	8.19	128	128
		2x	0x 1E	4.06	4.06	2.03	15.87	128	256
		4x	0x 3E	8.26	8.26	4.13	32.26	128	512
		8x	0x 7C	16.38	16.38	8.19	64.00	128	1024
	0	1x	0x 1E	4.06	2.03	2.03	7.94	256	256
		2x	0x 3E	8.26	4.13	4.13	16.13	256	512
		4x	0x 7C	16.38	8.19	8.19	32.00	256	1024
9 bits	1	1x	0x 1E	4.06	4.06	2.03	7.94	256	256
		2x	0x 3E	8.26	8.26	4.13	16.13	256	512
		4x	0x 7C	16.38	16.38	8.19	32.00	256	1024
	0	1x	0x 3E	8.26	4.13	4.13	8.06	512	512
		2x	0x 7C	16.38	8.19	8.19	16.00	512	1024
10 bits	1	1x	0x 3E	8.26	8.26	4.13	8.06	512	512
		2x	**0x 7C**	**16.38**	**16.38**	**8.19**	**16.00**	**512**	**1024**
	0	1x	0x 7C	16.38	8.19	8.19	8.00	1024	1024

Example 3–2. 10-kHz Sampling Rate

10 kHz Nominal Synthesis Rate									
32.768 kHz Oscillator Reference									
DAC Precision	**IntGenCtrl PDMCD Bit**	**Over-Sampling Factor**	**ClkSpdCtrl PLLM Register Value (hex)**	**Master Clock Rate (MHz)**	**PDM RATE (MHZ)**	**CPU Clock Rate (MHz)**	**Output Sampling Rate (kHz)**	**Number of Instructs Between DAC Interrupts**	**Number of Instructs Between 10 kHz Interrupts**
8 bits	1	1x	0x 13	2.62	2.62	1.31	10.24	128	128
		2x	0x 26	5.11	5.11	2.56	19.97	128	256
		4x	0x 4D	10.22	10.22	5.11	39.94	128	512
		8x	0x 9B	20.45	20.45	10.22	79.87	128	1024
	0	1x	0x 26	5.11	2.56	2.56	9.98	256	256
		2x	0x 4D	10.22	5.11	5.11	19.97	256	512
		4x	0x 9B	20.45	10.22	10.22	39.94	256	1024
9 bits	1	1x	0x 26	5.11	5.11	2.56	9.98	256	256
		2x	0x 4D	10.22	10.22	5.11	19.97	256	512
		4x	0x 9B	20.45	20.45	10.22	39.94	256	1024
	0	1x	0x 4D	10.22	5.11	5.11	9.98	512	512
		2x	0x 9B	20.45	10.22	10.22	19.97	512	1024
10 bits	“1”	1x	0x 4D	10.22	10.22	5.11	9.98	512	512
		2x	**0x 9B**	**20.45**	**20.45**	**10.22**	**19.97**	**512**	**1024**
	“0”	1x	0x 9B	20.45	10.22	10.22	9.98	1024	1024

3.3 Comparator

The MSP50C6xx provides a simple comparator that is enabled by a control register option. The inputs of the comparator are shared with pins PD_4 and PD_5. PD_5 is the noninverting input to the comparator, and PD_4 is the inverting input.

When the comparator is enabled, the conditional operation COND2 (normally associated with PD_1) becomes associated with the comparator result. In addition, the interrupts associated with PD_4 and PD_5 (namely, INT6 and INT7), become interrupts based on a *transition* in the comparator result. Finally, the start/stop function of TIMER1 may be controlled, indirectly, by a comparator transition. When enabled, the comparator controls the following four events:

(1) Steady-State Comparator TRUE	$V_{PD5} > V_{PD4}$	COND2 = TRUE . . .	
CIN2	has its conditional call taken.	JIN2	has its conditional jump taken.
CNIN2	has its conditional call ignored.	JNIN2	has its conditional jump ignored.
(2) Steady-State Comparator FALSE	$V_{PD5} < V_{PD4}$	COND2 = FALSE . . .	
CIN2	has its conditional call ignored.	JIN2	has its conditional jump ignored.
CNIN2	has its conditional call taken.	JNIN2	has its conditional jump taken.
(3) Comparator transition FALSE-to-TRUE		V_{PD5} rises above V_{PD4} . . .	
INT6 trigger event (If interrupt mask bit, D4, is set)			
TIMER1 stops counting (If INT7 flag was set and TIMER1 ENABLE was cleared)			
(4) Comparator transition TRUE-to-FALSE		V_{PD5} falls below V_{PD4} . . .	
INT7 trigger event (If interrupt mask bit, D5, is set)			
TIMER1 starts counting (If INT6 flag was cleared and TIMER1 ENABLE was cleared)			

With regards to the transition events, the rising-edge in the comparator is a trigger for INT6. This happens independently of any activity associated with TIMER1. TIMER1, on the other hand, can be stopped by a rising edge of the comparator. The INT7 flag must be set, and the TIMER1 ENABLE must be cleared before the event.

INT6 flag refers to bit 6 within the interrupt flag register (IFR, peripheral port 0x39). This bit is automatically SET anytime that an INT6 event occurs. This causes the device to branch to the INT6 vector if the associated mask bit is set (IntGenCtrl, address 0x38, bit 6). The INT6 flag is automatically CLEARed when the device branches to the INT6 vector at 0x7FF6. Refer to Section 2.7, *Interrupt Logic*, for more details)

The INT6 Flag may also be SET or CLEARed deliberately, at any time, in software. Use the OUT instruction with the associated I/O port address (IFR, address 0x39).

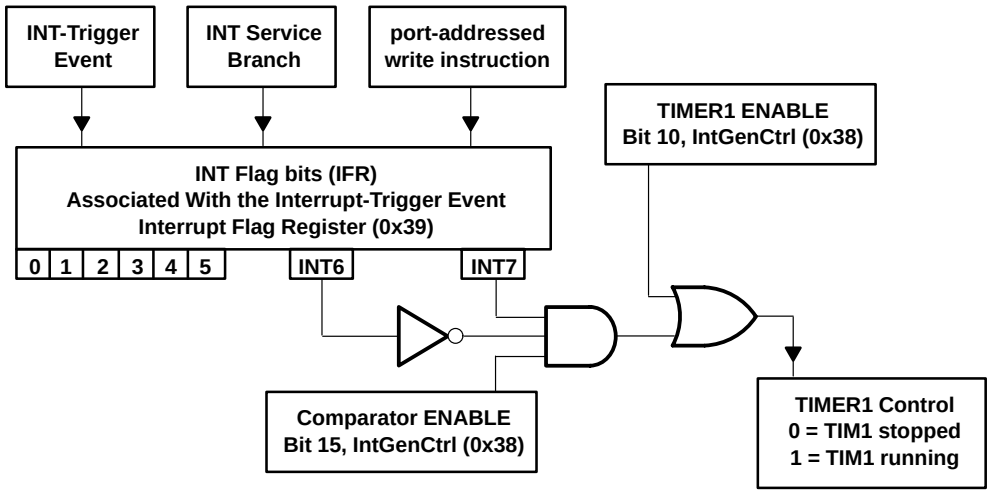
INT7 flag refers to bit 7 within the interrupt flag register. This bit is automatically SET anytime that an INT7 event occurs. This causes the device to branch to the INT7 vector if the associated mask bit is set (IntGenCtrl, address 0x38, bit 7). The INT7 flag is automatically cleared when the device branches to the INT7 vector at 0x7FF7.

The INT7 Flag may also be SET or CLEARed at any time, in software. Use the OUT instruction with the associated I/O port address (IFR, address 0x39).

The TIMER1 enable bit is set or cleared in software: bit 10 of the IntGenCtrl.

Similarly, the *falling*-edge event in the comparator is a trigger for INT7. This happens independently of any activity associated with TIMER1. TIMER1 can be started by the falling-edge of the comparator. The INT6 flag must be cleared, and the TIMER1 ENABLE must be cleared before the event.

Figure 3–2. Relationship Between Comparator/Interrupt Activity and the TIMER1 Control



The comparator, along with all of its associated functions, is enabled by setting bit 15 of the interrupt/general control register (IntGenCtrl, address 0x38). The default value of the register is zero: comparator disabled.

Note: IntGenCtrl Register Bit 15

At the time that bit 15 in the IntGenCtrl is set, PD_4 and PD_5 become the comparator inputs. At any time during which bit 15 is set, PD_4 and PD_5 MUST be set to INPUT (I/O Port D Control, address 0x1C, bits 4 and 5 CLEARed). Failure to do so may result in a bus contention.

The function of pins PD_4 and PD_5, and the behavior of events COND2, INT6, INT7, and TIMER1 are different, depending on whether the comparator has been enabled or disabled. A summary of the various states appears in the following table:

Comparator ENABLED	SET bit 15 in the IntGenCtrl, address 0x38 . . .
PD_4 functions as comparator negative input	(port D Control, 0x1C, bit 4 MUST be 0)
PD_5 functions as comparator positive input	(port D Control, 0x1C, bit 5 MUST be 0)
COND2 maps to the state of the comparator	(PD_5 relative to PD_4)
INT6 is triggered by PD_5 rising above PD_4	(IntGenCtrl, 0x38, bit 6 must be 1)
INT7 is triggered by PD_5 falling below PD_4	(IntGenCtrl, 0x38, bit 7 must be 1)
TIMER1 may be started by PD_5 rising above PD_4	(assuming TIMER1 Enable is 0 and INT6 flag is 0)
TIMER1 will be stopped by PD_5 falling below PD_4	(assuming TIMER1 Enable is 0 and INT7 flag is 1)
Comparator DISABLED	**CLEAR bit 15 in the IntGenCtrl, address 0x38 . . .**
PD_4 functions as a general-purpose I/O pin	(See Section 3.1.1)
PD_5 functions as a general-purpose I/O pin	(See Section 3.1.1)
COND2 maps to the state of the I/O pin PD_1	(See Section 3.1.4)
INT6 is triggered by a rising edge at PD_4	(IntGenCtrl, 0x38, bit 6 must be 1)
INT7 is triggered by a falling edge at PD_5	(IntGenCtrl, 0x38, bit 7 must be 1)
TIMER1 is started/stopped in software by setting/clearing TIMER1 enable (IntGenCtrl, 0x38, bit 10)	

3.4 Interrupt/General Control Register

The interrupt/general control (IntGenCtrl) is a 16-bit wide port-mapped register located at address 0x38. The primary component in the IntGenCtrl is the 8-bit interrupt mask register (IMR). The IMR is used to individually enable all interrupts except RESET. Each bit of the IMR is associated with one of the interrupts described in Section 3.1.5. An interrupt is enabled when the appropriate IMR bit is set. The IMR is located at bits 0 through 7 in the IntGenCtrl. Bit 0 is associated with INT0, which is the highest priority interrupt. Bit 7 is associated with INT7. Refer to Section 2.7, *Interrupt Logic*, for more information regarding the interrupt-system logic and initialization sequence.

IntGenCtrl register

address 0x38 (16-bit wide location)

15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
CE	AR	PD	EP	E2	E1	S2	S1	D5	D4	PF	D3	D2	T2	T1	DA
								low priority							high priority

0x0000 : State after RESET low

CE : Comparator enable
AR : ARM bit
PD : Pulse-density clock: PDMCD
EP : Enable pullup resistors on port F

E2 : Enable TIMER2 (1 value starts timer)
E1 : Enable TIMER 1 (1 value starts timer)
S2 : Clock source for TIMER2 (0 = MC/2, 1 = ref Osc)
S1 : Clock source for TIMER1 (0 = MC/2, 1 = ref Osc)

Interrupt mask register

D5 : port D_5 falling-edge
D4 : port D_4 rising-edge
PF : any port F falling-edge
D3 : port D_3 falling-edge
D2 : port D_2 rising-edge
T2 : TIMER2 underflow
T1 : TIMER1 underflow
DA : DAC timer underflow

(1 value enables interrupt service)

The remaining bits in the IntGenCtrl have various control functions which are not directly related to the interrupt system. Four of these are related to the timer functions. Bits 8 and 9 are used to select the clock sources which govern the rates of TIMER1 and TIMER2. Clearing bit 8 chooses 1/2 MC as the source for TIMER1 (i.e., the TIMER runs at one-half the frequency of the Master Clock). Setting bit 8 chooses the reference oscillator (RTO or CRO) as the source for TIMER1. (The same applies for bit 9 and TIMER2.) Bits 10 and 11 are used to enable TIMER1 and TIMER2, respectively. Setting bit 10 *starts* TIMER1, and clearing bit 10 *stops* TIMER1. (The same applies for bit 11 and TIMER2).

The upper four bits in the IntGenCtrl have independent functions. Bit 12 is the enable bit for the pull-up resistors on port F. Setting this bit applies individual pull-up resistors to each of the F port pins (see Section 3.1.2, *Dedicated Input Port F*).

Bit 13 is the PDMCD bit for the pulse-density modulation clock. Clearing this bit yields a PDM clock rate equal to one-half the frequency of the master clock (i.e., the CPU clock rate). Setting bit 13 yields a PDM rate equal to the rate of the master clock (see Section 3.2.3, *PDM Clock Divider*)

Bit 14 is the ARM bit. If the master clock has been suspended during sleep, then the ARM bit must be set (before the IDLE instruction), in order to allow a programmable interrupt to wake the MSP50C6xx. Refer to Section 2.11, *Reduced Power Modes*, for more information.

Finally, the top-most bit in the IntGenCtrl is the comparator enable bit. Setting bit 15 enables the comparator and all of its associated functions. Some of the MSP50C6xx's conditions, interrupts, and timers behave differently, depending on whether the comparator is enabled or disabled by this bit. Refer to Section 3.3, *Comparator*, for a full description.

3.5 Hardware Initialization States

The RESET pin is configured at all times as an external interrupt. It provides for a hardware initialization of the MSP50C6xx. When the RESET pin is held low, the device assumes a deep sleep state and various control registers are initialized. After the RESET pin is taken high, the Program Counter is loaded with the value stored in the RESET Interrupt Vector.

Note: Internal Power Reset Function

There is no power-on reset function internal to the MSP50C6xx. After the initial power-up or after an interruption in power, the RESET pin must be cycled low-to-high. The application circuitry must therefore provide a mechanism for accomplishing this during a power-up transition or after a power fluctuation.

The application circuits shown in Section 6.1, *Application Circuits*, illustrate one implementation of a reset-on-power-up circuit. The circuit consists of an RC network (100 kΩ, 1 μF). When powering V_{DD} from 0 V, the circuit provides some delay on the RESET pin's low-to-high transition. This delay helps to ensure that the MSP50C6xx initialization occurs *after* the power supply has had time to stabilize between V_{DD} MIN and V_{DD} MAX. V_{DD} MIN and V_{DD} MAX are the minimum and maximum supply voltages as rated for the device. The circuit shown, however, may not shield the RESET pin from *every* kind of rapid fluctuation in the power supply. At any time that the power supply falls below V_{DD} MIN, even momentarily, then the RESET pin must be held low and then high once again, either by the user of the device or by some other external circuitry (refer to the MSP50C6xx data sheet (SPSS023), *Electrical Specifications* section).

When the RESET pin is held low, the MSP50C6xx is considered reset and has the following internal states:

RESET low . . .

- ❑ I/O ports are be placed in a high impedance Input condition: Ports A, B, C, D, and E.
- ❑ All outputs on Port G is are set to low (0x0000).
- ❑ Device is placed in a deep sleep state.
- ❑ PLL circuitry, master clock, CPU clock, and TIMERs are stopped.
- ❑ Current draw from the V_{DD} is less than 10 μA in this condition.
- ❑ Interrupt flag register (IFR at address 0x39) is *not* automatically cleared.
- ❑ Internal RAM is *not* automatically cleared.

Note: Internal RAM State after Reset

The RESET low will not change the state of the internal RAM, assuming there is no interruption in power. This applies also to the interrupt flag register. The same applies to the states of the accumulators in the computational unit.

When RESET is brought back high again, many of the programmable controls and registers are left in their default states:

RESET high, just after low . . .

- ❑ No reference oscillator is enabled. PLL runs at its minimum achievable rate.
- ❑ Master clock runs at a very slow frequency (less than 100 kHz).
- ❑ PLL multiplier is set to 0x00 (renders slowest speed for MC, once reference is enabled).
- ❑ RTO oscillator trim bits are set to zero (renders slowest speed for RTO, once enabled).
- ❑ Interrupt mask register is 0x00. Global interrupt enable is clear. All Interrupts are disabled.
- ❑ I/O Ports A through E and output Port G have the same state as in RESET low.
- ❑ All pull-up resistors on input Port F are disabled.
- ❑ DAC circuitry is disabled (no PDM pulsing).
- ❑ Both TIMER1 and TIMER2 are disabled. Count-down and period registers are 0x0000.
- ❑ The status register is *partially* initialized, as specified in Table 3–2.
- ❑ Idle state clock control and ARM bit are both set to zero.

- ❑ The processor begins by executing the following steps:

  1) ROM block protection word is read from address 0x7FFE.

  2) ROM block protection word is loaded to an internal register.

  3) RESET interrupt vector is read from address 0x7FFF.

  4) Program counter is loaded with the value read from (3); execution resumes there.

Note: Stack Pointer Initialization

The software stack pointer (R7) must be initialized by the programmer, so that it points to some legitimate address in data memory (RAM). This must be done prior to any CALL or Ccc instruction. If this is not done, then the first push/pop operation performed will use the current location pointed to by R7.

Table 3–2. State of the Status Register (17 bit) after RESET Low-to-High (Bits 5 through 16 are left uninitialized)

Bit	Bit Name	Initialized Value	Description
0	XM	0	Extended sign mode disabled
1	UM	0	Unsigned multiplier mode disabled (allows signed multiplier mode)
2	OM	0	Overflow mode disabled (allows ALU normal mode)
3	FM	0	Shift mode for fractional multiplication disabled (allows unsigned fractional/integer arithmetic)
4	IM	0	Global interrupt enable bit
5	(reserved)	Same state as before RESET	Reserved for future use
6	XZF		Transfer equal-to-zero status bit
7	XSF		Transfer sign status bit
8	RCF		Auxiliary register carry-out status bit
9	RZF		Auxiliary register equal-to-zero status bit
10	OF		Accumulator overflow status bit
11	SF		Accumulator sign status bit (extended 17th bit)
12	ZF		Accumulator equal-to-zero status bit (16 bits)
13	CF		Accumulator carry-out status bit (16th ALU bit)
14	TF1		Test flag 1
15	TF2		Test flag 2
16	TAG		Memory tag

Chapter 4

Assembly Language Instructions

This chapter describes in detail about MSP50P614/MSP50C614 assembly language. Instruction classes, addressing modes, instruction encoding and explanation of each instruction is described.

Topic **Page**

4.1	Introduction	4–2
4.2	System Registers	4–2
4.3	Instruction Syntax and Addressing Modes	4–8
4.4	Instruction Classification	4–22
4.5	Bit, Byte, Word and String Addressing	4–44
4.6	MSP50P614/MSP50C614 Computational Modes	4–49
4.7	Hardware Loop Instructions	4–53
4.8	String Instructions	4–55
4.9	Lookup Instructions	4–57
4.10	Input/Output Instructions	4–59
4.11	Special Filter Instructions	4–59
4.12	Conditionals	4–69
4.13	Legend	4–70
4.14	Individual Instruction Descriptions	4–74
4.15	Instruction Set Encoding	4–189
4.16	Instruction Set Summary	4–198

4.1 Introduction

In this chapter each MSP50P614/MSP50C614 class of instructions is explained in detail with examples and restrictions. Most instructions can individually address bits, bytes, words or strings of words or bytes. Usable program memory is 30K by 17-bit wide and the entire 17-bits are used for instruction set encoding. The execution of programs can only be executed from internal program memory. Usable program memory starts from location 800h. The data memory is 640 by 17-bits of static RAM, 16 bits of which are an arithmetic value. The 17th bit is used for flags or tags.

4.2 System Registers

A functional description of each system register is described below.

4.2.1 Multiplier Register (MR)

The multiplier uses this 16-bit register to multiply with the multiplicand. MOV instructions are used to load the MR register. The multiplicand is usually the operand of the multiply instructions. All multiply, multiply-accumulate instructions, and filter instructions (FIR, FIRK, COR and CORK) use the MR register (see Section 4.11 for detail).

4.2.2 Shift Value Register (SV)

The shift value register is 4-bits wide. For barrel shift instructions, the multiplier operand decodes a 4-bit value in the shift value register (SV) to a 16-bit value. For example, a value of 7H in the SV register is decoded to a multiplier operand of 0000000010000000 binary. In effect, this causes a left shift of 7 bits to in the final 32-bit product. In other words, a nonzero value, say *k* ($0 \leq k \leq 15$), in the SV register means padding *k* number of zeros to the right of the final result.

4.2.3 Data Pointer Register (DP)

The data pointer register (DP) is a 16-bit register that is used to point to a program memory location for various look up table instructions. DP is not directly loaded by the user, It is loaded during the execution of lookup instructions overwriting the previous content of the DP register. Lookup instructions are described in detail in Section 4.9. The DP register auto-increments the next logical program memory location after the execution of a lookup instruction. In addition to lookup instructions, the filter instructions FIRK and CORK (see Section 4.11 for detail) use the DP pointer to look up filter coefficients. It may be required to context save and restore the DP in interrupt service routines.

4.2.4 Program Counter (PC)

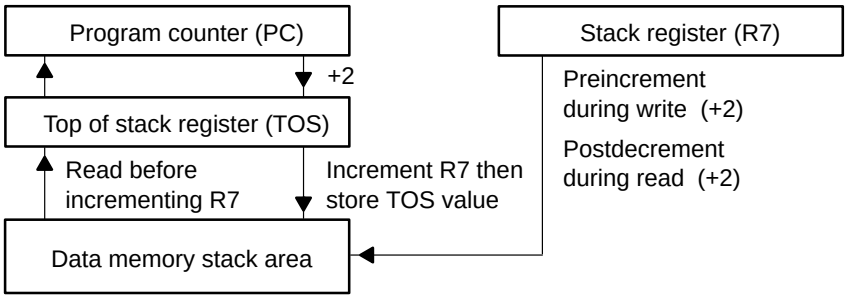
The program counter (PC) holds the program memory location to be used for the next instruction's execution. It increments (by 1 for single word instructions

or by 2 for double word instructions) each execution cycle and points to the next program memory location to fetch. During a maskable interrupt, the next PC address is stored in the TOS register and is reloaded from TOS after the interrupt encounters an IRET instruction. Call and jump instructions also store the next instruction address by adding PC+2 and then storing the result in the TOS register. Upon encountering a RET instruction, the TOS value is reloaded to the PC. Call instructions may not precede RET instructions. Similarly, a RET instruction may not immediately follow another RET instruction. In these conditions, pipeline operations breaks down and the PC never recovers its return address from the TOS register. The processor stalls, and the only solution is to reset the device. On the other hand, RET can be safely replaced by IRET eliminating processor stalls in all conditions. However, IRET takes one more cycle than RET.

4.2.5 Top of Stack, (TOS)

The top of stack (TOS) register holds the value of the stack pointed by the stack register (R7). The MSP50P614/MSP50C614 hardware uses TOS register for very efficient returns from CALL instructions. Figure 4-1 shows the operation of the TOS register. When call instructions are executed, the old TOS register value is pushed into the stack by pre-incrementing R7. The current PC value is incremented by 2 to compute the final return address and is then stored in the TOS register. Thus, the TOS register holds the next PC value pointing to the next instruction. When the subroutine reaches the RET instruction, the program counter (PC) is loaded with the TOS register. Next, the TOS is loaded with the value pointed to by R7. Finally, the stack register (R7) is decremented.

Figure 4–1. Top of Stack (TOS) Register Operation



The MSP50P614/MSP50C614 development tools use the TOS register for parameter passing. The TOS register must be used with caution inside user programs. If the TOS register and stack register (R7) are not restored to their previous values after using the TOS register in an application, the program can hang the processor or cause the program to behave in an unpredictable way.

It is recommended to avoid using the TOS register altogether in applications and leave its operation to development tools only.

4.2.6 Product High Register (PH)

This register holds the upper 16 bits of the 32-bit result of a multiplication, multiply-accumulate, or shift operation. The lower 16 bits of the result are stored in the PL register. The PH register can be loaded directly by MOV instructions. Special move accumulate instructions MOVAPH, MOVAPHS, MOVSPH, MOVSPHS also use the PH register.

4.2.7 Product Low Register (PL)

This register holds the lower 16 bits of the 32-bit result of a multiplication, multiply-accumulate, or shift operation. The upper 16 bits of the result are stored in the PH register. There are no instructions that load or save the PL register directly, but multiply-accumulate instructions allow the contents of the PL register to be added, subtracted or transferred to the accumulator.

4.2.8 Accumulators (AC0–AC31)

There are 32 accumulators on the MSP50P614/MSP50C614. Each is 16 bits wide. The first sixteen accumulators, AC0–AC15, have offset accumulators, AC16–AC31, and vice versa. At any one time, four accumulators can be selected through accumulator pointer registers, AP0–AP3 (see section 4.2.9). Some instructions can specify offset accumulators which are the accumulators pointed to by AP*n* +16 or AP*n* –16 (whichever is in the range 0 to 31). The offset accumulators are indicated by an offset bit (A~) in some instructions. When this bit is 0, A*n* points to the accumulator directly. If it is 1, then A*n*~ points to the offset (for some instructions this scheme changes). The selected accumulator pointer register should contain the index to the corresponding accumulator. For example, if AP0 has a value of 25, then it is pointing to accumulator AC25. If the offset bit is 1, A0~, then it is pointing to accumulator AC9 (25–16=9). Because, accumulators can only be addressed through accumulator pointers, special symbols are used in MSP50P614/ MSP50C614 instructions. Accumulators are indicated by the symbol A*n*, where *n* ranges from 0 to 3. The symbol indicates that the accumulator pointed to by AP*n* is the referring accumulator. If AP*n* has a value of *k*, it is pointing to accumulator AC*k*. Similarly, A*n*~ points to the offset accumulator pointed by AP*n*. For example, if AP3 = 22, then A3 is accumulator AC22 and A3~ is accumulator AC6.

During accumulator read operations, both A*n* and offset A*n*~ are fetched. Depending on the instruction, either or both registers may be used. In addition, some write operations allow either register to be selected.

The accumulator block can also be used in string operations. The selected accumulator (A*n* or A*n*~) is the least significant word (LSW) of the string and is restored at the end of the operation. String instructions are described in detail in section 4.8.

4.2.9 Accumulator Pointers (AP0–AP3)

The accumulator pointer (AP) registers are 5-bit registers which point to one of the 32 available accumulators. The APs contain the index of accumulators. Many instructions allow preincrement or predecrement accumulator pointers. Such instructions have a suffix of ++A for preincrement or – –A for predecrement. Accumulator pointers can be stored or loaded from memory using various addressing modes. Limited arithmetic operations can be performed on accumulator pointers.

Bit	Bits 16 – 5	4	3	2	1	0
AP0–AP3	Not used	Points to **A***n* n = val (b0–b4)				

4.2.10 Indirect Register (R0–R7)

Indirect registers, R0–R7, are 16-bit registers that are used in various addressing modes or as general-purpose registers. R0, R1, R2 and R3 can be used solely as general-purpose registers. These registers can also be used as indirect registers with relative addressing.

The R4 or LOOP register is used with instructions BEGLOOP and ENDLOOP to define a hardware controlled loop. If R4 is loaded with a value, *n* ($0 \leq n \leq 32767$), the BEGLOOP and ENDLOOP block will be executed *n+2* times. The loop stops when R4 becomes negative.

The R5 or INDEX register is used with indirect addressing and relative addressing modes of certain instructions.

The R6 or PAGE register is used with page relative addressing and relative flag addressing.

The R7 or STACK register holds the pointer to the stack. It can be used as a general-purpose register as long as no CALL/RET instructions are used before restoring it with its old value. However, this register can only be used as a general-purpose register when maskable interrupts are disabled. The old

value of the STACK register should be stored before use and restored after use. This register must point to the beginning of the stack in the RESET initialization routine before any CALL instruction or maskable interrupts can be used. CALL instructions increment R7 by 2., RET instructions decrement R7 by 2. The stack in MSP50P614/MSP50C614 is positively incremented.

4.2.11 String Register (STR)

The string register (STR) holds the length of the string used by all string instructions. MOV instructions are used to load this register to define the length of a string. The value in this register is not altered after the execution of a string instruction. A value of zero in this register defines a string length of 2. Thus, a numerical value, n_S, in the STR register, defines a string length of n_S+2. The maximum string length is 32. Therefore, $0 \leq n_S \leq 30$ corresponds to actual string lengths from 2 to 32.

4.2.12 Status Register (STAT)

The status register (STAT) provides the storage of various single bit mode conditions and condition bits. As shown in Table 4–1, mode bits reside in the first 5 LSBs of the status register and can be independently set or reset with specific instructions. See section 4.6 for detail about these computational modes. Condition bits and flags are used for conditional branches, calls, and flag instructions. Flags and status condition bits are stored in the upper 10 bits of the 17-bit status register. MOV instructions provide the means for context saves and restores of the status register. The STAT should be initialized to 0000h after the processor resets.

The XSF and XZF flags are related to data flow to or from the internal data bus. If the destination of the transfer is an accumulator, then the SF, ZF, CF and OF flags are affected. If the destination of the transfer is R*x*, the RCF and RZF flags are affected. If the destination of the transfer is through the internal databus, the XSF and XZF flags are affected. The SF flag is the sign flag and it is equal to the most significant bit of an accumulator when an accumulator instruction is executed. ZF is the zero flag and is set when the instruction causes the accumulator value to become zero. CF is the carry flag and is set when the instruction causes a carry. A carry is generated by addition, subtraction, multiplication, multiply-accumulate, compare, shifting and some MOV instructions (that have accumulation features). CF is reset if no carry occurs after execution of an instruction. OF is set when a computation causes overflow in the result. It is reset if no overflow occurs during an accumulator based instruction. Overflow saturation mode is set by the OM bit as explained in Section 4.6.

Table 4–1. Status Register (STAT)

Bit	Name	Function
0	XM	Sign extended mode bit. This bit is one, if sign extension mode is enabled. See MSP50P614/MSP50C614 Computational Modes, Section 4.6.
1	UM	Unsigned multiplier mode. This bit is one if unsigned multiplier mode is enabled. See MSP50P614/MSP50C614 Computational Modes, Section 4.6.
2	OM	Overflow mode. This bit is one if overflow (saturation) mode is enabled. See MSP50P614/MSP50C614 Computational Modes, Section 4.6.
3	FM	Fractional multiplication shift mode. This bit is set if fractional mode is enabled. See MSP50P614/MSP50C614 Computational Modes, Section 4.6.
4	IM	Maskable interrupt enable mode. If this bit is zero, all maskable interrupts are disabled.
5	Reserved	Reserved for future use.
6	XZF	Transfer(x) equal to zero status (flag) bit. In transfer instructions, this bit is set if the operation cause the destination result to become zero (excluding accumulator and R*x* registers).
7	XSF	Transfer(x) sign status (flag) bit. In transfer instructions, the sign bit of the value is copied to this bit if the destination is not accumulator or R*x* registers.
8	RCF	Indirect register carry out status (flag) bit. This bit is set if an addition to the value of R*x* register caused a carry.
9	RZF	Indirect register equal to zero status (flag) bit. This bit is set if the R*x* register content used by the instruction is zero.
10	OF	Accumulator overflow status (flag) bit. This bit is set if an overflow occurs during computation in ALU.
11	SF	Accumulator sign status (flag) bit (extended 17th bit). This bit is set if the 16^{th} bit (the sign bit) of the destination accumulator is 1.
12	ZF	Accumulator equal to zero status (flag) bit (16 bits). This bit is set to 1 if the result of previous instruction cause the destination accumulator to become zero.
13	CF	Accumulator carry out status (flag) bit (16^{th} ALU bit).
14	TF1	Test Flag 1. Test flags are related with Class 8 instructions discussed later.
15	TF2	Test Flag 2. Test flags are related with Class 8 instructions discussed later.
16	TAG	Memory tag. Holds the 17^{th} bit whenever a memory value is read.

4.3 Instruction Syntax and Addressing Modes

MSP50P614/MSP50C614 instructions can perform multiple operations per instruction. Many instructions may have multiple source arguments. They can premodify register values and can have only one destination. The addressing mode is part of the source and destination arguments. In the following subsection, a detail of the MSP50P614/MSP50C614 instruction syntax is explained followed by the subsection which describes addressing modes.

4.3.1 MSP50P614/MSP50C614 Instruction Syntax

All MSP50P614/MSP50C614 instructions with multiple arguments have the following syntax:

name [*dest*] [, *src*] [, *src1*] [, *mod*]

where the symbols are described as follows:

name	*name* of the instruction. Instruction names are shown in bold letters. If the instruction *name* is followed by a B, the arguments are all byte types. If *name* is followed by an S, all arguments are word string (strings of words) types. If *name* is followed by BS, all arguments are byte string types.
dest	*dest*ination of data to be stored after the execution of an instruction. Optional or not used for some instructions. Destination is also used as both a source and a destination for some instructions. If a destination is specified, it must always be the first argument. Destinations can be system registers or data memory locations referred by addressing modes. This is instruction specific.
src	*sourc*e of first data. Optional or not used for some instruction. Source can be a system register, a data memory location referred by addressing modes, or a program memory location. This is instruction specific.
src1	*sourc*e of second data. Some instructions use a second data source. Optional or not used for some instructions. Source 1 can be a system register, a data memory location referred by addressing modes, or a program memory location. This is instruction specific.
mod	pre or post modification of a register. The meaning of *mod* is instruction specific.
[ ]	Square brackets represent optional arguments. Some instructions have many combinations of source and destination registers and addressing modes. The combination is instruction class specific.

The possible combinations of sources, destinations and modifications are dependent on the instruction class. Instruction classes are discussed in detail in Section 4.4.

4.3.2 Addressing Modes

The addressing modes on the MSP50P614/MSP50C614 are immediate, direct, indirect with post modification, and three relative modes. The relative modes are:

- Relative to the INDEX or R5 register. The effective address is (indirect register + INDEX).
- Short relative to the PAGE or R6 register. The effective address is (PAGE+7-bit positive offset).
- Long relative to R*x*. The effective address is (indirect register R*x* + 16-bit positive offset).

When string instructions are executed, the operation of the addressing mode used is modified. For all addressing modes except indirect with post modification, a temporary copy of the memory address is used to fetch the least significant data word of the string. Over the next *n* instruction cycles, the temporary copy of the address is auto-incremented to fetch the next *n* words of the string. Since the modification of the address is temporary, all R*x* registers are unchanged and still have reference to the least significant data word in memory. String data fetches using the *indirect with post modification* addressing mode and writes the modified address back to the indirect register at each cycle of the string. This will leave the address in the R*x* register pointing to the data word whose address is one beyond the most significant word of the string.

All addressing modes except immediate addressing are encoded in bits 0 to 7 of the instruction's op-code. Table 4–2 through Table 4–6 show the encoding of various addressing modes. Addressing mode bits (except immediate and flag addressing) come with an *am*, R*x* and *pm* field. These are combined into a single field called {*adrs*}. The appropriate decoding and syntax for each addressing mode with the {*adrs*} field is described in Table 4–4. The *pm* field only applies to indirect addressing. For other addressing modes, it is coded as zero.

Table 4–2. Addressing Mode Encoding

Bit	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode						*next A*				*am*			R*x*			*pm*	

am contains addressing mode bits 5 – 7. See Table 4–4 for details.
R*x* is the register being used. See for Table 4–3 for details.
pm is the post modification flag. See Table 4–3 for details.
next A is the accumulator pointer premodification field. See Table 4–5 for details.

Table 4–3. Rx Bit Description

R*x*			Operation
0	0	0	R0
0	0	1	R1
0	1	0	R2
0	1	1	R3
1	0	0	R4 or LOOP
1	0	1	R5 or INDEX
1	1	0	R6 or PAGE
1	1	1	R7 or STACK

Table 4–4. Addressing Mode Bits and {adrs} Field Description

Relative Addressing Modes	**Clocks** *clk*	**Words** *w*	**Repeat Operation**‡ *clk*	{*adrs*}	**addressing mode encoding, *adrs***							
					7	6	5	4	3	2	1	0
					am			R*x* (x = 0 … 7)			*pm*	
Direct†	2	2	n_R+4	**dma16*	0	0	0	R*x*			0	0
Short relative	1	1	n_R+2	*R6 + *offset7*	1	offset7						
Relative to R5	1	1	n_R+2	*R*x* + R5	0	1	0	R*x*			0	0
Long relative†	2	2	n_R+4	*R*x* + *offset16*	0	0	1	R*x*			0	0
Indirect	1	1	n_R+2	*R*x*	0	1	1	R*x*			0	0
				*R*x*++							0	1
				*R*x*––							1	0
				*R*x*++R5							1	1

† = *dma16* and *offset16* is the second word
‡ n_R is RPT instruction argument

Table 4–5. MSP50P614/MSP50C614 Addressing Modes Summary

ADDRESSING	SYNTAX	OPERATION
Direct	***name*** *[dest,] [src,] *dma16 [*2] [, next A]* ***name*** *******dma16 [*2] [,src] [, next A]*	Second word operand (*dma16*) used directly as memory address.
Long Relative	***name*** *[dest] [,src] ,*R*x+offset16 [, next A]* ***name*** ******Rx+*offset16 [,src] [, next A]*	Selects one of 8 address registers as base value and adds the value in the second word operand. Does not modify the base address register.
Relative to R5 (INDEX)	***name*** *[dest] [,src] ,*R*x+R5 [, next A]* ***name*** *R*x**+**R5 *[,src] [, next A]*	Selects one of 8 address registers as base value and adds the value in R5. Does not modify the base address register.
Indirect	***name*** *[dest] [, src] ,*Rx*++R5 [, next A]* ***name*** *[dest] [, src] ,*Rx *[, next A]* ***name*** *[dest] [, src] ,*Rx++ [, next A]* ***name*** *[dest] [, src]* ***,****Rx— [, next A]* ***name*** *Rx++R5 *[, src] [, next A]* ***name*** *Rx *[, src] [, next A]* ***name*** *Rx++ *[, src] [, next A]* ***name*** *Rx— *[, src] [, next A]*	Selects one of 8 address registers to be used as the address, post modifications of increment, decrement, and + INDEX(R5) are possible.
Short Relative	***name*** *[dest] [, src] ,*R6+offset7 [, next A]* ***name*** *R6+*offset7 [, src] [, next A]*	Selects PAGE(R6) register as the base address and adds a 7-bit positive address offset from operand field (b6–b0). This permits the relative addressing of 128 bytes or 64 words. Does not modify the PAGE address register. *k* is shown as constant.
Global Flag	***name*** TF*n, dma6* ***name*** *dma6, TFn*	For use with flag instructions only. Adds lower 7 bits of instruction to a fixed address base reference of zero. 64 fixed flags are addressed by this mode beginning at address 0000h.
Relative Flag	***name*** TF*n,* *R6+*offset6* ***name*** *R6+*offset6*, TF*n*	For use with flag instructions only. Adds lower 7 bits of instruction(lsb set to zero) to a address base reference stored in the PAGE register (R6). 64 flags relative to PAGE may be addressed with this mode.

Table 4–6. Auto Increment and Auto Decrement Modes

Operation	Syntax	*next A*	
No modification		0	0
Aufto decrement	—A	0	1
Auto increment	++ A	1	0
String mode		1	1

Table 4–6 describes the accumulator pointer auto preincrement or predecrement syntax. Not all instructions can premodify accumulator pointers. The *next A* field is a two-bit field using bits 10 and 11 of only certain classes of instructions. Instructions with a [*next A*] have either a —A or a ++A in the instruction. See Table 4–6.

For any particular addressing mode, replace the {*adrs*} with the syntax shown in Table 4–4. To encode the instruction, replace the *am*, R*x* and *pm* bits with the bits required by the addressing mode (Table 4–4). For example, the instruction MOV A*n*[~]**,** {*adrs*} [, *next A*] indicates all of the following (only partial combinations are shown):

```
MOV  A0, *0xab12          ; n = 0, {adrs} = dma16 = 0xab12
MOV  A1, *R6+0x2f, ++A    ; n = 1, {adrs} = *R6+0x2f, offset7 = 0x2f,
                            [next A] = ++A
MOV  A2~, *R0+R5, ––A     ; n = 2, {adrs} = *R0+R5, x = 0, [next A] = ––A
MOV  A3, *R1+0x12ef       ; n = 3, {adrs} = *R1+0x12ef, x = 1,
                            offset16 = 0x12ef
MOV  A0, *R2              ; n = 0, {adrs} = *R2, x = 2
MOV  A1, *R3++, ––A       ; n = 1, {adrs} = *R3++, x = 3, [next A] = —A
MOV  A2~, *R4––           ; n = 2, {adrs} = *R4––, x = 4
MOV  A3, *R7++R5, ++A     ; n = 3, {adrs} = *R7++R5, x = 7, [next A] = ++A
```

Flag instructions apply to certain classes of instructions (Class 8a). They address only the flag bit by either a 6-bit global address or a 6-bit relative address from the indirect register R6. If bit 0 of these instructions is 0, then bits 1 to 6 of the opcode are taken as the bit address starting from data memory location 0000h. If bit 0 is 1, then bits 1 to 6 are used as an offset from the page register R6 to compute the relative address. Bits 0 to 6 of flag instructions are written as {*flagadrs*} throughout this manual. When this symbol appears, it should be replaced by the syntax and bits shown in Table 4–7

For example, AND TF*n***,** {*flagadrs*} can be written as follows (not all possible combinations are shown):

```
AND TF1, *0x21      ; global flag addressing, flag address is 0x21 absolute
AND TF2, *R6+0x21   ; relative flag addressing, flag address is R6+0x21
                      absolute
```

Table 4–7. Flag Addressing Field {flagadrs} for Certain Flag Instructions (Class 8a)

Flag Addressing Modes	**Clocks** *clk*	**Words** *w*	**Repeat Operation,**† *clk*	{*flagadrs*} **Syntax**	**flag addressing mode encoding, *flagadrs***						
					6	5	4	3	2	1	0
					flag address bits						g/r
Global	1	1	n_R+2	**dma6*	dma6						0
Relative	1	1	n_R+2	*R6+*offset6*	offset6						1

† n_R is RPT argument

4.3.3 Immediate Addressing

The address of the memory location is encoded in the instruction word or the word following the opcode is the immediate value. Single word instructions take one clock cycle and double word instructions take two clock cycles.

Syntax:

name dest, [*src*,] *imm* [, *next A*]

Where: *imm* is the immediate value of a 16-bit number.

Example 4.3.1 `ADD AP0, 0x1A`
Assume the initial processor state in Table 4–8 before execution of this instruction. This instruction adds the immediate value 0x1A to AP0. Final result AP0 = 0x1A + 2 = 0x1C.

Table 4–8. Initial Processor State for the Examples Before Execution of Instruction

Registers (*register#* = *value*)			
AP0 = 2	AP1 = 21 (0x15)	AP2 = 11 (0x0B)	AP3 = 29 (0x1D)
R0 = 0x0454	R1 = 0x0200	R2 = 0x0540	R3 = 0x03E2
R4 = 0x0000	R5 = 2	R6 = 0x03E4	R7 = 0x0100
AC2 = 0x13F0	AC1 = 0x0007	AC17 = 0x0112	AC20 = 0x3321
AC3 = 0xFEED	AC28 = 0x11A2	AC29 = 0xAB	AC19 = 0x1200
MR = 0x1A15			
data memory (**address* = *data*) [word address; to convert to byte, *address* multiply by 2]			
*0x022A = 0x0400	*0x01F2 = 0x12AC	*0x02A1 = 0x1001	*0x012F = 0x0000
*0x0100 = 0x0ABC	*0x0080 = 0x0000	*0x0001 = 0x499A	*0x01FA = 0x0112
program memory (**address* = *data*)			
*0x13F0 = 0x1B12			

Example 4.3.2 `MOV R5, 0xF000`
Loads the immediate value 0xF000 to R5 register. Final result, R5 = 0xF000.

Example 4.3.3 `MOVB MR, 0xF2`
Loads the immediate byte 0xf2 to MR register. Final result, MR = 0xf2.

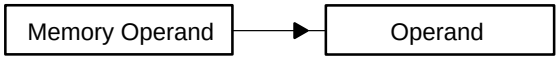
Example 4.3.4 `AND A0, A0~, 0xFF20, --A`
Assume the initial processor state in Table 4–8 before execution of this instruction. The source accumulator pointer AP0 is predecremented. After predecrement, A0 points to AC1**,** and A0~ points to AC17**.** AC17 is anded with the immediate 16-bit value (0xFF20) and the result is stored in AC1. Final result, AP0 = 1, AC1 = 0xFF20 AND AC17 = 0xFF20 AND 0x0112 = 0x0100.

4.3.4 Direct Addressing

Direct addressing always requires two instruction words. The second word operand is used directly as the memory address. The memory operand may be a label or an expression.

Syntax:

name [*dest*,] [*src*,] ***dma16* [* 2] [, *next A*]
name **dma16* [* 2] [, *src*] [, *next A*]



Note the multiplication by 2 with the data memory address. This only needs to be done for word addresses, i.e., the address that points to 16-bit words. This is not required for byte addresses. This is explained in detail in section 4.5.

Example 4.3.5 `MOV A2, *0x022A * 2`
Refer to the initial processor state in Table 4–8 before execution of this instruction. Loads the contents of data memory location 0x022A (=0x0400) to A2 or AC11. The MSP50P614/MSP50C614 always accesses data memory as byte addresses. To read a word address, multiply the address by 2. Final result, A2 = AC11 = 0x0400.

Example 4.3.6 `MOV A1~, *0x01F2 * 2, ++A`
Refer to the initial processor state in Table 4–8 before execution of this instruction. Preincrement AP1. After preincrement A1 is AC22 and A1~ is AC6. The content of data memory location 0x01F2 (=0x12AC) is then loaded to accumulator AC22 (offset of AC6). Final result, AP1=22, AC6 = 0x12AC.

Example 4.3.7 `SUB A1~, A1, *0x02A1 * 2, --A`
Refer to the initial processor state in Table 4–8 before execution of this instruction. Predecrement AP1. After predecrement A1 is AC20 and A1~ is AC4. Subtract the content of 0x02A1 (=0x1001) in data memory from AC20 and store result to AC4. Final result, AP1 = 20, AC4 = AC20 – 0x1001 = 0x3321 – 0x1001 = 0x2320.

Example 4.3.8 `MOV *0x012F * 2, *A0`
Refer to the initial processor state in Table 4–8 before execution of this instruction. This is a table lookup instruction. This instruction reads the program memory address stored in A0 or AC2 and stores the data in data memory location 0x012F. Final result, *0x012F = 0x1B12.

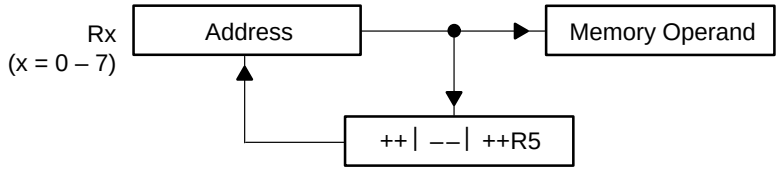
Example 4.3.9 `MULR *0x02A1 * 2`
Refer to the initial processor state in Table 4–8 before execution of this instruction. Multiply MR with the contents of 0x02A1. The MSB of the result is stored in PH register and rounded. The LSB is ignored. Final result, multiply MR • *0x02A1 = 0x1A15 • 0x1001 = 0x1A16A15, PH = 0x01A1.

4.3.5 Indirect Addressing

Indirect addressing uses one of 8 registers (R0...R7) to point memory addresses. The selected register can be post-modified. Modifications include increments, decrements, or increments by the value in the index register (R5). For post-modifications, the register increments or decrements itself by 2 for word operands and by 1 for byte operands. Syntaxes are shown in Table 4–9.

Table 4–9. Indirect Addressing Syntax

Syntax	Operation
*name [dest,] [src,] ,*R*x*++R5 [*, next A]* *name* *R*x*++R5 [*, src] [, next A]*	Premodify accumulator pointer if *next A* is included. Add R*x* with R5.
*name [dest,] [src,] ,*R*x [, next A]* *name* *R*x [, src] [, next A]*	Premodify accumulator pointer if *next A* is included. Use address pointed by R*x*, R*x* content unchanged
*name [dest,] [src,] ,*R*x++ [, next A]* *name* *Rx++* [, *src*] [, *next A*]	Premodify accumulator pointer if *next A* is included. Use address pointed by R*x*, post increment R*x* after use
*name [dest,] [src,] ,*R*x–– [, next A]* *name* *R*x–– [, src] [, next A]*	Premodify accumulator pointer if *next A* is included. Use address pointed by R*x*, post decrement R*x* after use



Note that the R*x* registers treats data memory as a series of bytes. Therefore, when a word is loaded, R*x*++ increments by 2 (R*x*–– decrements by 2). When loading a word address into R*x*, the address must be converted into a byte address (by multiplying by 2). For example, if we want R*x* to point to the word address, 0x100, R*x* should be loaded with 0x100*2=0x200.

Example 4.3.10 `MOV A1~, *R1++R5, ++A`

Refer to the initial processor state in Table 4–8 before execution of this instruction. Preincrement AP1. After preincrement A1 is AC22 and A1~ is AC6. The contents of the data memory location stored in R1 are loaded into accumulator AC6. R1 is then incremented by R5. Final result, AP1=22, AC6 = 0xacb, R1 = R1 + R5 = 0x0202. Note that the addressing of the R*x* registers is byte addressing.

Example 4.3.11 `ADD A3~, A3, R6++R5, --A`

Refer to the initial processor state in Table 4–8 before execution of this instruction. Predecrement AP3. After predecrement, A3 is AC28 and A3~ is AC12. The contents of the data memory location stored in R6 are added to AC28. The result is stored in accumulator AC12. R6 is then incremented by R5. Final result, AP3=28, AC12 = AC28 + *R6 = 0x11A2 + 0x12AC = 0x244E, R6 = R6+R5 = 0x3E6. Note that the R*x* registers use byte addresses.

Example 4.3.12 `MOV *R5++R5, A0~, ++A`
Refer to the initial processor state in Table 4–8 before execution of this instruction. Preincrement AP0. After preincrement, A0 is AC3 and A0~ is AC19. The contents of AC19 are stored in the data memory location in R5. R5 is then incremented by R5. Final result, AP0=3, R5 = 0x0004, *0x0002 = 0xFEED.

Example 4.3.13 `MOV A2, *R0`
Refer to the initial processor state in Table 4–8 before execution of this instruction. The contents of the data memory address in R0 are loaded into A2 (AC11). Final result, AC11 = 0x0400. Note the addressing is byte addressing. Thus, *R0 = 0x0454 indicates the word memory location 0x454/2 = 0x022A.

Example 4.3.14 `IN *R4++, 0x00`
The contents of the I/O port location 0x00 (port PPA) are stored in the location pointed to by R4. R4 is incremented by 2 after this operation.

Example 4.3.15 `MOVB *R7++, A3`
Refer to the initial processor state in Table 4–8 before execution of this instruction. Store the lower 8 bits of A3 (AC29) in the data memory byte address pointed to by R7. R7 is then incremented by one. Notice that to find the word address, divide the address in R7 by 2. Final result, R7=0x0101, *0x0100 = 0xAB (byte address) or *0x80 = 0xAB00 (word address).

Example 4.3.16 `OUT 0x08, *R1--`
Refer to the initial processor state in Table 4–8 before execution of this instruction. The contents of the data memory byte location stored in R1 are placed on port 0x08 (port PPB). R1 is then decremented by 2. Final result, R1 = 0x01FE, *0x08 = 0xCB. Port PPB is 8-bits wide, so the upper 8-bits of *R1 (0x0A) are ignored.

4.3.6 Relative Addressing

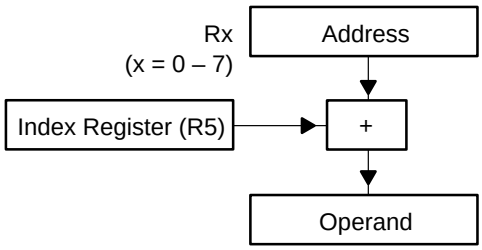
There are three types of relative addressing on the MSP50P614/MSP50C614: short relative, long relative, and relative to the index register, R5. These addressing modes are described below.

4.3.6.1 Relative to Index Register R5

This relative addressing mode uses one of the 8 address registers (R0–R7) as a base value. The index register, R5, is added to the base address value in R*x.* The base address register is not modified. Thus, the effective address is R*x* + R5.

Syntax:

name [*dest*,] [*src*,] *R*x*+R5 [, *next A*]
name *R*x*+R5 [, *src*] [, *next A*]



Example 4.3.17 `AND A0, *R3+R5`

Refer to the initial processor state in Table 4–8 before execution of this instruction. A0 is accumulator AC2. The contents of the data memory byte location pointed to by R3+R5 is ANDed with AC2. The result is stored in AC2. The values in R3 and R5 are unchanged. Final result, AC2 = AC2 AND *0x01F2 = 0x13F0 AND 0x12AC = 0x12A0.

Example 4.3.18 `MOV *R2+R5, A2~, ++A`

Refer to the initial processor state in Table 4–8 before execution of this instruction. Preincrement AP2. After preincrement, A2 is AC12 and A2~ is AC28. Store AC28 in the data memory byte location R2+R5. The values in R2 and R5 are unchanged. Final result, *0x02A1 = 0x11A2.

Example 4.3.19 `ADD A0~, A0, *R4+R5, --A`

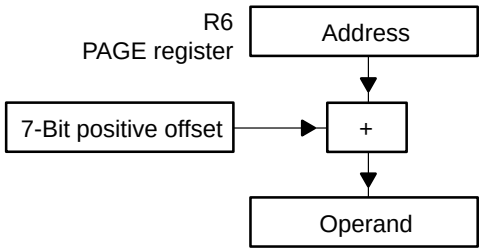
Refer to the initial processor state in Table 4–8 before execution of this instruction. Predecrement AP0. After predecrement, A0 is AC1 and A0~ is AC17. Add AC1 to the contents of byte location R4+R5 and put the result in AC17. The values in R4 and R5 are unchanged. Final result, AC17 = AC1 + *(R4+R5) = 0x0007 + *0x0002 = 0x0007 + 0x499A = 0x49A1.

4.3.6.2 Short Relative

Short relative (also called PAGE Relative) addressing selects the Page register (R6) as a base value and adds a 7-bit positive offset from the operand. The page register is not modified.

Syntax:

name [*dest*,] [*src*,] *R6+*offset7* [, *next A*]
name *R6+*offset7* [, *src*] [, *next A*]



Example 4.3.20 `MOV A3, *R6+0x10`
Refer to the initial processor state in Table 4–8 before execution of this instruction. Load A3 (AC29) with the contents of byte address, R6+0x10. The value of R6 is unchanged. Final result, AC29=0x0112.

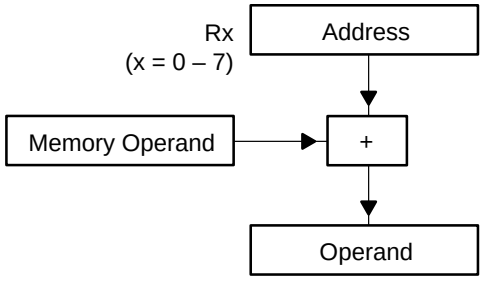
Example 4.3.21 `ADD A0~, A0, *R6+0x10, ++A`
Refer to the initial processor state in Table 4–8 before execution of this instruction. Preincrement AP0. After preincrement, A0 is AC3 and A0~ is AC19. Add AC3 to the contents of byte address R6+0x10 and store the result in AC19. The value in R6 is unchanged. Final result, AC19 = AC3 + *(R6+0x10) = 0xFEED + *0x01FA = 0xFEED + 0x0112 = 0xFFFF.

4.3.6.3 Long Relative

Long relative addressing selects one of the 8 address registers (R*x*) as a base value and adds the value of the second word operand. The base address register is not modified.

Syntax:

name [*dest*,] [*src*,] *Rx+*offset16* [, *next A*]
name *Rx+*offset16* [, *src*] [, *next A*]



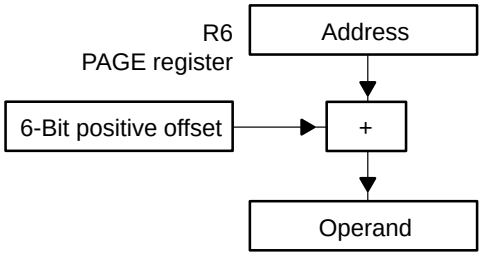
Example 4.3.22 `MOV A0~, *R1+0x0254, ++A`
Refer to the initial processor state in Table 4–8 before execution of this instruction. Preincrement A0. After preincrement, A0 is AC3 and A0~ is AC19. Load the contents of the data memory byte location R1+0x0254 into AC19. R1 remains unchanged. Final result, AP0=3, AC19=*(R1+0x0254) = *0x022A = 0x0400.

Example 4.3.23 `MOV *R7+0x0442, MR`
Refer to the initial processor state in Table 4–8 before execution of this instruction. Store the value in MR to data memory byte location, R7+0x0442. R7 remains unchanged. Final result, *0x02A1 = 0x1A15.

4.3.7 Flag Addressing

This addressing mode addresses only the 17th bit (the flag/tag bit) located in data memory. This addressing applies to Class 8a instructions as explained in section 4.4. Using flag addressing, the flag bit can be loaded or saved. In addition, various logical operations can be performed without affecting the remaining 16 bits of the selected word. Two addressing modes are provided. The first addressing mode, global flag addressing, has bit 0 set to zero and a six bit field (b1–b6) that defines the flag word address. The second mode, relative flag addressing, has bit 0 set to one and a 6-bit field (b1–b6) that defines the flag address relative to R6 (see Figure 4–2). In other words, the, i.e., effective address = (contents of R6) + (6-bit offset). In flag addressing, R6 contains the address that points to the 17th bit. This should not be confused with byte addresses and word addresses.

Figure 4–2. Relative Flag Addressing



Syntax: name {*dest*}, {*src*}

Global Flag: *name* TF*n*, *dma6*

name dma6, TF*n*

Relative Flag: *name* TF*n*, *R6+*offset6*

name *R6+offset6, TFn

Example 4.3.24 `MOV *0x02, TF2`

Take the test flag 2 bit (TF2 in the status register) and place it into the 17th bit of the data memory location 0x02.

Example 4.3.25 `AND TF1, *0x20`

AND the test flag 1 bit (TF1 in status register) with the 17th bit of the data memory location 0x20 and store the result in the TF1 bit of the STAT.

Example 4.3.26 `OR TF2, *R6+0x02`

OR the test flag 2 bit (TF2 in status register) with the 17th bit of the data memory location *(R6+0x02) and store the result in the TF2 bit in of the status register. So, if R6=0x0100, then relative flag address is 0x0102.

Example 4.3.27 `XOR TF1, *R6+0x20`

XOR the test flag 1 bit (TF1 in status register) with the 17th bit of the data memory location *(R6+0x20) and store the result in TF1 bit of the status register. So, if R6=0x0100, then relative flag address is 0x0120.

4.3.8 Tag/Flag Bits

The words TAG and flag may be used interchangeably in this manual. The TAG bit is the 17th bit of a word of data memory. There are 640 words of RAM, each 17 bits wide, on the C614. Therefore, there are 640 TAG bits on the C614. When an instruction of the format,

```
MOV accumulator, RAM
```

is performed, the STAT register is affected by various properties of this transfer. The TAG bit of the RAM location is copied into the TAG bit of the STAT register during such transfers.

The TAG bit can be modified using several instructions: STAG**,** RTAG**,** SFLAG**,** RFLAG. There are subtle differences between these instructions that the user must understand before using them. The first difference between the xTAG and xFLAG instructions is the addressing.

```
STAG *0x0000       ;sets the TAG bit of RAM word zero
RTAG *0x0002       ;clears the TAG bit of RAM word one
STAG  *0x0002 * 2  ;sets the TAG bit of RAM word two
```

STAG and RTAG use RAM byte addresses to specify which TAG to set or clear. This immediately causes confusion since there are 1280 bytes and only 640 TAGs. What happens when an odd byte is used to set a tag with STAG?

```
STAG *0x0001       ;sets the TAG bit of RAM word zero
STAG *0x0003       ;sets the TAG bit of RAM word one
STAG  *0x0005 * 2  ;sets the TAG bit of RAM word five
```

All word boundaries in RAM start at even numbers, RAM_{even}. If an odd byte, $RAM_{even} + 1$ is used to set a TAG, then the TAG for RAM_{even} is set. Thus,

```
STAG *0x0000
STAG *0x0001
```

are functionally equivalent.

As a sharp contrast, the **SFLAG** and **RFLAG** instructions use RAM word addresses to specify which TAG to set or clear.

```
SFLAG *0x0000      ;sets the TAG bit of RAM word zero
SFLAG *0x0001      ;sets the TAG bit of RAM word one
```

Another difference between the xTAG and xFLAG instructions is the addressing modes. STAG and RTAG can use {adrs} addressing modes. This includes, direct, short relative, relative to R5, long relative, and indirect addressing modes. This affects the number of clock cycles it takes to execute xTAG instructions.

However, xFLAG instructions use {flagadrs} addressing modes. This includes global (dma6) and relative (R6 + 6–bit offset). Both take only one clock cycle.

Possible sources of confusion: Consider the following code,

```
ram0   equ0x0000 *2  ;RAM word zero
ram1   equ0x0001 *2  ;RAM word one
ram2   equ0x0002 *2  ;RAM word two
STAG   *ram1
MOV    A0,*ram1      ;TAG bit is set in STAT register
RTAG   *ram1
SFLAG  *ram1         ;This sets the TAG bit of ram2!
MOV    A0,*ram1      ;TAG bit is not set in STAT register!
MOV    TF1,*ram1     ;TF1 bit in STAT is set!?
```

Explanation: The first three instructions perform as you would expect. The TAG bit is set at the RAM variable, ram1. The TAG bit is set in the STAT register when the MOV instruction executes. Finally, ram1's TAG bit is cleared.

The next two instructions are problematic. When SFLAG sets the tag bit, it will set the tag bit for the second word location, ram2. This does not set the TAG bit for ram1. What is worse is that the value in ram1 must be less than 64 (dma6) since this is global addressing for SFLAG**.** To access TAG bits for higher RAM, the R6 (PAGE) register is needed.

The last instruction is also confusing. Why is TF1 set in the STAT even though ram1's TAG bit is not set? The answer is that this MOV instruction considers the {src} argument to be a word value instead of the usual byte value. Thus, this MOV instruction operates on ram2 rather than on ram1.

4.4 Instruction Classification

The machine level instruction set is divided into a number of classes. The classes are primarily divided according to field references associated with memory, hardware registers, and control fields. The following descriptions give class-encode bit assignments, the OP code value within the class, and the abbreviated field descriptions.

Some of the following symbols will be used repeatedly throughout this chapter as shown in Table 4–10 (for additional information see section 4.13).

Table 4–10. Symbols and Explanation

Symbol	Explanation
!	Invert the bit of the source. Used with flag addressing only.
{*adrs*}$_n$	The contents of the effective data memory address referred to by the addressing mode syntax. If n is specified, n bits are involved. If unspecified, data is 16 bits. See Table 4–4.
{*cc*}	Condition code mnemonic used with conditional branch/calls and test flag/bit instructions. Curly braces indicate this field is not optional.
{*flagadrs*}	Flag addressing syntax as shown in Table 4–7.
~A	Select offset accumulator as the destination accumulator if this bit is 1.
~A~	Can be either ~A or A~ based on the opcode (or instruction).
A~	Select offset accumulator as source if this bit is 1.
adrs	Addressing mode bits *am*, R***x***, *pm.* See Table 4–4.
A*n*	Accumulator pointed to by AP*n*. Accumulators cannot be referenced directly. For example, A22 is not valid since accumulators are only addressible though the accumulator pointers AP0–AP3. Therefore, to access accumulators, use A0, A1, A2 and A3. This should not be confused with AP*n* where AP is an *accumulator pointer*, not an accumulator.
A*n*~	Indicates the offset of the accumulator pointed to by accumulator pointer A*n*. This is also an accumulator, not an accumulator pointer.
Ap*n*	Accumulator pointer AP*n* where *n* = 0, 1, 2 or 3. The difference between A*n* and AP*n* is that A*n* is the accumulator pointed to by AP*n*. In both cases, *n* ranges from 0 to 3.
cc	Condition code bits used with conditional branch/calls and test flag/bit instructions.
clk	Clock cycles to execute the instruction
dma[n]	n bit data memory address. For example, dma8 means 8–bit location data memory address. If n is not specified, defaults to dma16.
flagadrs	Flag addressing bits as shown in Table 4–7.
flg	Test flag bit.
g/r	Global/relative flag bit for flag addressing.
imm[n]	n bit immediate value
k0...kn	Constant field bits.

Table 4–11. Symbols and Explanation (Continued)

Symbol	Explanation
next A	Accumulator control bits as described in Table 4–6.
[*next A*]	The preincrement (++A) or predecrement (––A) operation on accumulator pointers A*n* or A*n*~.
Not	NOT condition on conditional jumps, conditional calls or test flag instructions.
n_R	Value in the repeat counter loaded by repeat instruction.
n_S	Value in string register STR.
offset[n]	n bit offset from a reference register.
pma[n]	n bit program memory address. For example, pma8 means 8-bit program memory address. If n is not specified, defaults to pma16.
port[n]	n bit I/O port address.
R	R*x* registers are treated as general-purpose registers. These bits are not related to any addressing modes.
R*x*	Indirect register bits as described in Table 4–3.
s	Represents string mode if 1, otherwise normal mode.
x	Don't care

Instructions on the MSP50P614/MSP50C614 are classified based on the operations the instruction group performs (see Table 4–11). Each instruction group is referred to as a class. There are 9 instruction classes. Classes are subdivided into subclasses. Classes and opcode definitions are shown in Table 4–11.

Table 4–11. Instruction Classification

Class	Sub-Class	Description
1		Accumulator and memory reference instructions
	A	Accumulator and memory references with or without string operations and accumulator preincrementing
	B	Accumulator and memory references with or without string operations
2		Accumulator constant reference
	A	Short constant to accumulator
	B	Long constant to accumulator
3		Accumulator reference instructions with no addressing modes

Table 4–11. Instruction Classification (Continued)

Class	Sub-Class	Description
4		Register and memory reference
	A	Memory references that use R*x*; all addressing modes available
	B	Memory references with short constant fields operating on R*x*
	C	Memory references with long constant fields operating on R*x* in errata, has not been connected
	D	Memory references with R5 operating on R*x*
5		General mMemory reference instructions
6		I/O port and memory reference instructions
	A	Port/memory reference
	B	Port/accumulator reference
7		Program control instructions
	A	Macro call instructions
	B	Conditional and unconditional jump instructions
	C	Conditional and unconditional call instructions
8		Logical bit instructions
	A	Logical flag instructions
	B	Test status instructions
9		Miscellaneous instructions
	A	Filter instructions
	B	Miscellaneous short constant instructions
	C	Accumulator address instructions
	D	Other instructions

Table 4–12. Classes and Opcode Definition

Bit	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Class 1a	0	0	C1a		~A~	next A		A*n*		am			R*x*			pm	
Class 1b	0	1	C1b				s	A*n*		am			R*x*			pm	
Class 2a	1	0	1	0	C2a			A*n*		imm8							
Class 2b	1	1	1	0	0	next A		A*n*		C2b			0	0	1	A~	~A
Class 3	1	1	1	0	0	next A		A*n*		C3					0	A~†	~A
Class 4a	1	1	1	1	0	C4a	R			am			R*x*			pm	
Class 4b	1	0	1	1	C4b		k4	k3	k2	k7	k6	k5	R			k1	k0
Class 4c	1	1	1	1	1	1	1	0	0	0	C4c		R			x	x
Class 4d	1	1	1	1	1	1	1	0	0	1	C4d		R			x	x
Class 5	1	1	0	1	C5					am			R*x*			pm	
Class 6a	1	1	0	0	C6a	port4				am			R*x*			pm	
Class 6b	1	1	1	0	1	1	s	A***n***		port6						C6b	~A~
Class 7a	1	1	1	1	1	1	1	0	1	vector8							
Class 7b	1	0	0	0	0	0	Not	cc					rx			pm	
JMP *A*n*	1	0	0	0	1	0	x	A*n*		x	x	x	x	x	x	x	x
Class 7c	1	0	0	0	0	1	Not	cc					x	x	x	x	x
CALL *An	1	0	0	0	1	1	x	A*n*		x	x	x	x	x	x	x	x
Class 8a	1	0	0	1	1	flg	n	C8a		flagadrs							g/r
Class 8b	1	0	0	1	0	flg	Not	cc					R*x*			C8b	C8b
Class 9a	1	1	1	0	1	0	0	A*n*		C9a		0	R*x*			1	1
Class 9b	1	1	1	1	1	1	0	C9a		k							
Class 9c	1	1	1	1	1	0	1	A*n*		0	C9c	x	imm5				
Class 9d	1	1	1	1	1	1	1	1	0	C9d				0	0	0	0
ENDLOOP *n*	1	1	1	1	1	1	1	1	0	0	0	0	1	0	0	0	n
NOP	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

† Meaning of this bit depends on what class 3 instruction is used.

4.4.1 Class 1 Instructions: Memory and Accumulator Reference

This class of instructions controls execution between data memory and the accumulator block. In addition to the explicit opcode field that specifies an arithmetic operation, an eight-bit data memory addressing mode reference field (*am*, R*x*, *pm* i.e., *adrs* field) controls the addressing of one input operand, and a 4-bit field (A*n* and *next A* in class 1a) or 2-bit field (A*n* in class 1b) selects an accumulator location as the other input operand. The results are written to the addressed accumulator location (or to the offset accumulator in class 1a if ~A bit = 1). In addition, each instruction can be treated as a single word length operation or as a string, depending on the string control encoded in the op code (*s* = 1 in class 1b and A*n* = 11 binary in class 1a).

Class 1a provides the four basic instructions of load, store, add, and subtract between accumulator and data memory. Either the accumulator or the offset accumulator (A~ bit dependent) can be stored in memory with the MOV instruction. The MOV instruction can load the accumulator (or its offset) depending on the ~A bit. The ADD or SUB instructions add or subtract memory from an accumulator register and save the results in the accumulator register (~A=0) or its offset (~A=1). Two of the four codes provided by the *next A* field will cause a pre–increment or a predecrement of the accumulator register pointer (AP) prior to execution. This preincrement is a permanent change to the referenced AP and further expands the use of the accumulator block as an efficient workspace. Preincrements and predecrements are not available in string mode

One of the four codes of the A*n* field (A*n* = 11 binary) will cause the instruction to be treated as a multicycle string instruction. This will not result in any permanent modification to the referenced AP.

Since there is no reference to offset accumulators in Class 1b instructions, the execution operates on memory and accumulators. All other modes of control (string, preincrement/predecrement AP, data memory addressing modes, etc.) are provided for logical, byte, multiply-accumulate, and barrel shift instructions.

Table 4–13. Class 1 Instruction Encoding

Bit	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Class 1a	0	0	C1a		~A~	*next A*		A*n*		*adrs*							
Class 1b	0	1	C1b				*s*	A*n*		*adrs*							

Table 4–14. Class 1a Instruction Description

C1a		Mnemonic	Description
0	0	ADD A*n*[~], A*n*, {*adrs*} [, *next A*] ADDS A*n*[~], A*n*, {*adrs*}	Add contents of data memory location referred by {*adrs*} to accumulator A*n* and store the results in the same accumulator A*n* (if ~A=0) or offset accumulator A*n*~ (~A=1). ALU status is modified.
0	1	SUB A*n*[~], A*n*, {*adrs*} [, *next A*] SUBS A*n*[~], A*n*, {*adrs*}	Subtract contents of data memory location referred by {*adrs*} from accumulator A*n* and store the results in the same accumulator A*n* (if ~A=0) or offset accumulator A*n*~ (~A=1). ALU status is modified.
1	0	MOV A*n*[~], {*adrs*} *[, next A]* MOVS A*n*[~], {*adrs*}	Load accumulator A*n* (~A=0) or offset accumulator A*n*~ (~A=1) from data memory location referred to {*adrs*}. ALU status is modified.
1	1	MOV {*adrs*}, A*n*[~] [, next A] MOVS {*adrs*}, A*n*[~]	Store accumulator (A~=0) or offset accumulator (A~=1) to data memory location referred to by addressing mode {*adrs*}. Transfer status is modified.

Table 4–15. Class 1b Instruction Description

C1b				Mnemonic	Description
0	0	0	0	OR A*n*, {*adrs*} ORS A*n*, {*adrs*}	Logical OR the contents of the data memory location in {*adrs*} and the selected accumulator. Result(s) stored in accumulator(s). ALU status is modified
0	0	0	1	AND A*n*, {*adrs*} ANDS A*n*, {*adrs*}	Logical AND the contents of the data memory location in {*adrs*} and the accumulator. Result(s) stored in accumulator(s). ALU status is modified
0	0	1	0	XOR A*n*, {*adrs*} XORS A*n*, {*adrs*}	Exclusive OR the contents of the data memory location in {*adrs*} and the accumulator. Result(s) stored in accumulator(s). ALU status is modified
0	0	1	1	MOVB A*n*, $\{adrs\}_8$ MOVBS A*n*, $\{adrs\}_8$	Load the contents of the data memory location in {*adrs*}and to the lower 8 bits of the accumulator. Zero fill the upper byte in the accumulator ALU status is modified.
0	1	0	0	MOVB $\{adrs\}_8$, A*n* MOVBS $\{adrs\}_8$, A*n*	Store the lower 8 bits of accumulator to the data memory location in {*adrs*}. The data byte is automatically routed to either the lower byte or upper byte in the 16-bit memory word based on the LSB of the address. Transfer status is modified.
0	1	0	1	Reserved	N/A
0	1	1	0	CMP A*n*, {*adrs*} CMPS A*n*, {*adrs*}	Store the arithmetic status of the contents of {*adrs*} subtracted from accumulator into the ALU status bits. The accumulator is not modified.
0	1	1	1	MOV {*adrs*} , *A*n* MOVS {*adrs*} , *A*n*	Look up the value stored in program memory addressed by the accumulator and store in the data memory location in {*adrs*}. Transfer status is modified .
1	0	0	0	MULTPL A*n*, {*adrs*} MULTPLS A*n*, {*adrs*}	Multiply the MR register by the contents of {*adrs*} and transfer the lower 16 bits of the result to the accumulator. Latch the upper 16 bits into the PH register. ALU status is modified.
1	0	0	1	MOVSPH A*n*, MR, {*adrs*} MOVSPHS A*n*, MR, {*adrs*}	Load the MR register in signed mode from the data memory location in {*adrs*}. In parallel, subtract the PH register from the accumulator. The string bit will string with the previous ALU status (CF, ZF) but it will not load the string counter (executes once). ALU status is modified.
1	0	1	0	MOVAPH A*n*, MR, {*adrs*} MOVAPHS A*n*, MR, {*adrs*}	Load the MR register in signed mode from the data memory location in {*adrs*}. In parallel, add the PH register to the accumulator. The string bit will string with the previous ALU status (CF, ZF) but it will not load the string counter (executes once). ALU status is modified.

Table 4–15. Class 1b Instruction Description (Continued)

C1b				Mnemonic	Description
1	0	1	1	MULAPL A*n*, {*adrs*} MULAPLS A*n*, {*adrs*}	Multiply the MR register by the addressing mode {*adrs*} and add the lower 16 bits of the product to the accumulator. Latch the upper 16 bits into the PH register. ALU status is modified.
1	1	0	0	SHLTPL A*n*, {*adrs*} SHLTPLS A*n*, {*adrs*}	Shift left n bits (SV reg). The 16-bit contents of the data memory location in {*adrs*} are shifted and placed in accumulator (string) An. Zeros fill from the right and either zeros or ones fill the left depending on the sign (assuming XSGM mode is set). Transfer the lower 16 bits to the accumulator and latch the upper 16 bits in PH. ALU status is modified.
1	1	0	1	SHLSPL A*n*, {*adrs*} SHLSPLS A*n*, {*adrs*}	Shift left n bits (SV reg). The contents of the data memory location in {*adrs*} are placed in a 32-bit result. Zeros fill from the right and either zeros or sign extended ones fill the left (if XSGM mode is set). Subtract the lower 16 bits from the accumulator and latch the upper 16 bits in PH. ALU status is modified.
1	1	1	0	SHLAPL A*n*, {*adrs*} SHLAPLS A*n*, {*adrs*}	Shift left n bits (SV reg). The contents of the data memory location in {*adrs*} are placed into a 32-bit result. Zeros fill the right and either zeros or sign extended ones fill the left (in XSGM mode). Add the lower 16 bits to the accumulator and latch the upper 16 bits in PH. ALU status is modified.
1	1	1	1	MULSPL A*n*, {*adrs*} MULSPLS A*n*, {*adrs*}	Multiply the MR register by the contents of {*adrs*} and subtract the lower 16 bits of the product from the accumulator. Latch the upper 16 bits into the PH register. ALU status is modified.

4.4.2 Class 2 Instructions: Accumulator and Constant Reference

These instructions provide the capability to reference short (8 bits) or long (16 bits or (n_S+2) * 16-bit string) constants stored in program memory and to execute arithmetic and logical operations between accumulator contents and these constants. Since the MSP50P614/MSP50C614 is a Harvard type processor, these instructions are necessary and distinct from the general class of memory reference instructions. Subclass 2a, listed belows include references between accumulator and short 8-bit constants. This class has the advantage of requiring only 1 instruction word to code and 1 instruction cycle to execute Thus is particularly useful for control variables such as loop counts, indexes, etc. The short constants also provide full capability for byte operations in a single instruction word.

Subclass 2b references accumulator and long constants from program memory (16 bits for non string constants and (n_S+2) * 16 bits for string constants). Class 2b instructions take 2 instruction words to code. The execution of these instructions is 2 instruction cycles when the long constant is a single word. The execution is n_S+2 execution cycles for n_S word string

constants. Long constants (16 bits) and long string constants differ in that references are made to constants in the second word of the two-word instruction word. References made to a single 16-bit integer constant are immediate. That is, the actual constant value follows the first word opcode in memory. For string constants, the second word reference to the constants is immediate-indirect which indicates that the second word is the address of the least significant word of the string constant. This definition allows all long string constants to be located in a table and permits the reference in the machine language listing to be consistent with those of shorter constants.

Table 4–16. Class 2 Instruction Encoding

Bit	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Class 2a	1	0	1	0	C2a			A*n*		*imm8*							
Class 2b	1	1	1	0	0	*next A*		A*n*		C2b			0	0	1	A~	~A

Table 4–17. Class 2a Instruction Description

C2a			Mnemonic	Description
0	0	0	ADDB A*n*, *imm8*	Add an 8-bit positive constant to the accumulator and store the result in the accumulator. ALU status is modified.
0	0	1	MOVB A*n*, *imm8*	Load an 8-bit positive constant into accumulator. ALU status is modified.
0	1	0	SUBB A*n*, *imm8*	Subtract 8-bit positive constant from accumulator and store result accumulator. ALU status modified.
0	1	1	CMPB A*n*, *imm8*	Modify ALU status with the result of 8-bit positive value subtracted from accumulator. Original accumulator value not modified.
1	0	0	ORB A*n*, *imm8*	Logical OR 8-bit positive constant with accumulator and store result to accumulator. ALU status modified.
1	0	1	ANDB A*n*, *imm8*	Logical AND 8-bit positive constant with accumulator. Store result to accumulator. ALU status modified.
1	1	0	XORB A*n*, *imm8*	Logical XOR 8-bit positive constant with accumulator. Store result to accumulator. ALU status modified.
1	1	1	MOVB MR, *imm8*	Load 8-bit constant to Multiplier register (MR). Does not change UM mode in status register but will zero fill the top 8 bits in MR register. No change in status.

Table 4–18. Class 2b Instruction Description

C2b			Mnemonic	Description
0	0	0	ADD A*n*[~], A*n*[~], *imm16 [, next A]* ADDS A*n*[~], A*n*[~], *pma16*	Add long constant to accumulator (or offset accumulator if A~=1) and store result to accumulator (~A=0) or offset accumulator (~A=1). ALU status modified.
0	0	1	MOV A*n*[~], *imm16* [, *next A*] MOVS A*n*[~], *pma16*	Load long constant to accumulator (~A=0 or 1). ALU status is modified.
0	1	0	SUB A*n*[~], A*n*[~], *imm16* [, *next A*] SUBS A*n*[~], A*n*[~], *pma16*	Subtract a long constant from the accumulator (A~=0 or 1). Store the result in accumulator (~A=0) or offset accumulator (~A=1). ALU status is modified.
0	1	1	CMP A*n*[~], *imm16* [, *next A*] CMPS A*n*[~], *pma16*	Modify ALU status by subtracting a long constant from accumulator (A~=0) or from offset accumulator (A~=1). Neither accumulator or offset accumulator is modified
1	0	0	OR A*n*[~], A*n*[~], *imm16* [, *next A*] ORS A*n*[~], An[~], *pma16*	Logical OR a long constant with accumulator (A~=0 or 1). Store the result in accumulator(~A=0) or offset accumulator (~A=1). ALU status is modified.
1	0	1	AND A*n*[~], A*n*[~], *imm16* [, *next A*] ANDS A*n*[~], A*n*[~], *pma16*	Logical AND a long constant with accumulator (A~=0 or 1).Store the result to accumulator(~A=0 or 1) . ALU status is modified.
1	1	0	XOR A*n*[~], A*n*[~], *imm16* [, *next A*] XORS A*n*[~], A*n*[~], *pma16*	Logical exclusive OR a long constant with accumulator (A~=0 or 1) Store the result to accumulator (~A=0 or 1). ALU status is modified.
1	1	1	MOV MR, *imm16* [, *next A*]	Load a long constant to MR in signed mode. No change in status.

4.4.3 Class 3 Instruction: Accumulator Reference

These instructions reference the accumulator and, in some instances, specific registers for transfers. Some instructions use a single accumulator operand and others use both the accumulator and the offset accumulator to perform operations between two accumulator values. The A~ bit in the instruction word reverses the sense of the addressed accumulator and the addressed offset accumulator. In general, if A~=1, the instruction uses the offset accumulator as the input operand on single accumulator operand instructions. It interchanges the arithmetic order (subtract, compare, multiply–accumulate, etc.) of the two operands when both are used. Exceptions to the rule are the instructions NEGAC[S], NOTAC[S], MULSPL[S], MULAPL[S], MULTPL[S], SHLSPL[S], SHLTPL[S] and SHLAPL[S], which use the reverse A~ control (A~=1 for accumulator, A~=0 for offset accumulator). The ~A bit in the instruction word controls the destination of the result to be the accumulator (~A=0) or the offset accumulator (~A=1).

In addition to basic accumulator arithmetic functions this class also includes an accumulator lookup instruction and several register transfer instructions

between the accumulator and the MR, SV, or PH register. As with all accumulator referenced instructions, string operations are possible as well as premodification of one of 4 indirectly referenced accumulator pointer registers (AP).

Table 4–19. Class 3 Instruction Encoding

Bit	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Class 3	1	1	1	0	0	*next A*		A*n*		C3					0	A~	~A

Table 4–20. Class 3 Instruction Description

C3					Mnemonic	Description
0	0	0	0	0	NEGAC A*n*[~], A*n*[~] [, *next A*] NEGACS A*n*[~], A*n*[~]	Store the 2's complement of the source accumulator (A~=0 or 1) to the destination accumulator (~A=0 or 1). ALU status is modified.
0	0	0	0	1	NOTAC A*n*[~], A*n*[~] [, *next A*] NOTACS A*n*[~], A*n*[~]	Place the 1's complement of the source accumulator (A~=0 or 1) into the destination accumulator (~A=0 or 1). ALU status is modified.
0	0	0	1	0	MOV A*n*[~], *A*n*[~] [, *next A*] MOVS A*n*[~], *A*n*[~]	Look up a value in program memory addressed by accumulator (A~=0 or 1). Place the lookup value into the accumulator (~A=0 or 1). The lookup address is post–incremented in the DP register. ALU status is modified based on the lookup value.
0	0	0	1	1	ZAC A*n*[~] [, *next A*] ZACS A*n*[~]	Zero accumulator (~A=0 or 1). ALU status is modified.

0	0	1	0	0	SUB A*n*[~], A*n*, A*n*~ [, *next A*]
					SUB A*n*[~], A*n*~, A*n* [, *next A*]
					SUBS A*n*[~], A*n*, A*n*~
					SUBS A*n*[~], A*n*~, A*n*
						Subtract offset accumulator from accumulator (A~=0) or subtract accumulator from offset accumulator (A~=1). Store the result in accumulator (~A=0 or 1). ALU status is modified.

0	0	1	0	1	ADD A*n*[~], A*n*~, A*n* [, *next A*] ADDS A*n*[~], A*n*~, A*n*	Add accumulator to offset accumulator and store result to accumulator (~A=0 or 1). ALU status is modified.
0	0	1	1	0	SHLAC A*n*[~], A*n*[~] [, *next A*] SHLACS A*n*[~], A*n*[~]	Shift accumulator left 1 bit and store the result into accumulator(~A=0) or offset accumulator (~A=1). The LSB is set to zero and the MSB is stored in a carryout status bit. ALU status is modified.
0	0	1	1	1	MOV A*n*, A*n*~ [, *next A*] MOVS A*n*, A*n*~	Copy accumulator (A~=0 or 1) to accumulator (~A=0 or 1). ALU status is modified.

† These instructions have a special 1 word string operations when string mode is selected. The instructions ignore the string count, executing only once but maintain the carry and comparison to zero operation of the previous arithmetic operation as if the sequence of the previous string instruction and this instruction execution was a part of a larger string operation.

Table 4–20. Class 3 Instruction Description (Continued)

C3					Mnemonic	Description
0	1	0	0	0	XOR A*n*[~], A*n*~, A*n* [, *next A*] XORS A*n*[~], A*n*~, A*n*	Logically exclusive OR accumulator with offset accumulator and store the results in accumulator (~A=0 or 1). ALU status is modified.
0	1	0	0	1	OR A*n*[~], A*n*~, A*n* [, *next A*] ORS A*n*[~], A*n*~, A*n*	Logically OR accumulator with offset accumulator and store results into accumulator (~A=0 or 1). ALU status is modified.
0	1	0	1	0	AND A*n*[~], A*n*~, A*n* [, *next A*] ANDS A*n*[~], A*n*~, A*n*	Logically AND accumulator with offset accumulator and store result(s) into accumulator (~A=0 or 1). ALU status is modified.
0	1	0	1	1	SHRAC A*n*[~], A*n*[~] [, *next A*] SHRACS A*n*[~], A*n*[~]	Shift accumulator or offset accumulator right 1 bit and store result in accumulator (~A=0 or 1). MSB will be set to zero or be set equal to the sign bit (XSGM dependent). ALU status is modified.
0	1	1	0	0	SUB A*n*[~], A*n*[~], PH [, *next A*] SUBS A*n*[~], A*n*[~], PH †	Subtract product high register from accumulator (A~=0) or from offset accumulator (A~=1) and store the result into accumulator (~A=0) or into the offset accumulator (~A=1). ALU status is modified. String bit causes subtract with carry status (CF).
0	1	1	0	1	ADD A*n*[~], A*n*[~], PH [, *next A*] ADDS A*n*[~], A*n*[~], PH †	Add product high register to accumulator or to offset accumulator and store the result into accumulator (~A=0 or 1). ALU status is modified. The string bit causes an add with carry status (CF).
0	1	1	1	0	MOV A*n*[~], PH [, *next A*] MOVS A*n*[~], PH †	Transfer product high register to accumulator (~A=0) or offset accumulator (~A=1). ALU status is modified. String bit will cause stringing with current ZF status bit.
0	1	1	1	1	EXTSGN A*n*[~] [, *next A*] EXTSGNS A*n*[~] †	Copy SF bit in status register to all 16 bits of the accumulator or offset accumulator. On strings, the accumulator address is preincremented causing the sign of the addressed accumulator to be extended into the next accumulator address.

1	0	0	0	0	CMP A*n*~, An [, *next A*]
					CMP A*n*, An~ [, *next A*]
					CMPS A*n*~, A*n*
					CMPS A*n*, A*n*~
						Subtract offset accumulator from accumulator(A~=0) or subtract accumulator from offset accumulator (A~=1) and store the status of the result into ALU status. Accumulator or offset accumulator original value remains unchanged.

1	0	0	0	1	reserved	N/A
1	0	0	1	0	reserved	N/A
1	0	0	1	1	reserved	N/A

† These instructions have a special 1 word string operations when string mode is selected. The instructions ignore the string count, executing only once, but maintain the carry and comparison to zero operation of the previous arithmetic operation as if the sequence of the previous string instruction and current instruction were part of a larger string operation.

Table 4–20. Class 3 Instruction Description (Continued)

C3					Mnemonic	Description
1	0	1	0	0	MOV SV, A*n*[~] [, *next A*] MOVS SV, A*n*[~]	Transfer accumulator(A~=0) or offset accumulator (A~=1) to SV register. Transfer status is modified.
1	0	1	0	1	MOV PH, A*n*[~] [, *next A*] MOVS PH, A*n*[~]	Transfer accumulator (A~=0) or offset accumulator (A~=1) to PH register. Transfer status is modified.
1	0	1	1	0	MOV MR, A*n*[~] [, *next A*] MOVS MR, A*n*[~]	Transfer accumulator (A~=0) or offset accumulator (A~=1) to MR register in the signed multiplier mode (UM bit in status register set to 0). Transfer status is modified.
1	0	1	1	1	MOVU MR, A*n*[~] [, *next A*]	Transfer accumulator (A~=0 or 1) to MR register in the unsigned multiplier mode(UM bit set to 1). Transfer status is modified.
1	1	0	0	0	MULSPL A*n*[~], A*n*[~] [, *next A*] MULSPLS A*n*[~], A*n*[~]	Multiply the MR register by accumulator (A~=1) or offset accumulator (A~=0) , subtract lower 16 bits of the product from the offset accumulator (A~=1) or accumulator (A~=0). Store in the accumulator (~A=0) or offset accumulator (~A=1). Latch the upper 16 bits in PH. ALU status is modified.
1	1	0	0	1	MULAPL A*n*[~], A*n*[~] [, *next A*] MULAPLS A*n*[~], A*n*[~]	Multiply MR register by accumulator (A~=1) or offset accumulator (A~=0) , add lower 16 bits of product to offset accumulator (A~=1) or accumulator (A~=0) and store to accumulator (~A=0) or offset accumulator (~A=1). Latch upper 16 bits in PH. ALU status is modified.
1	1	0	1	0	SHLTPL A*n*[~], A*n*[~][, *next A*] SHLTPLS A*n*[~], A*n*[~]	Barrel shift the accumulator (A~=1 or 1) value n bits left (SV reg). Store the upper 16 bits of the 32-bit shift result to PH (msbs extended by XM mode bit). Transfer the lower 16 bits to accumulator (~A=0) or offset(~A=1). ALU status is modified.
1	1	0	1	1	MULTPL A*n*[~], A*n*[~] [, *next A*] MULTPLS A*n*[~], A*n*[~]	Multiply MR register by accumulator(A~=1) or offset (A~=0), transfer lower 16 bits of product to accumulator (~A=0) or offset accumulator(~A=1). Latch upper 16 bits of Product to PH register. ALU status is modified.
1	1	1	0	0	SHLSPL A*n*[~], A*n*[~] [, *next A*] SHLSPLS A*n*[~], A*n*[~]	Barrel shift the accumulator(A~=1) or offset accumulator (A~=0) value n bits left (SV reg). Store the upper 16 bits to PH. Subtract the lower 16 bits of value from offset (A~=1) or accumulator (A~=0) and store in accumulator (~A=0) or offset accumulator (~A=1). ALU status is modified.
1	1	1	0	1	SHLAPL A*n*[~], A*n*[~] [, *next A*] SHLAPLS A*n*[~], A*n*[~]	Barrel shift the accumulator(A~=1) or offset accumulator (A~=0) value n bits left (SV reg). Store the upper 16 bits to PH. Add the lower 16 bits of value to offset accumulator (A~=1) or accumulator (A~=0) and store in accumulator (~A=0) or offset accumulator(~A=1). ALU status is modified.

Table 4–20. Class 3 Instruction Description (Continued)

C3					Mnemonic	Description
1	1	1	1	0	MUL A*n*[~] [, *next A*] MULS A*n*[~]	Multiply MR register by accumulator (A~=1) or offset accumulator (A~=0) and latch the rounded upper 16 bits of the resulting product into the PH register.
1	1	1	1	1	SHL A*n*[~] [, *next A*] SHLS A*n*[~]	Barrel shift the accumulator (A~=1) or offset accumulator (A~=0) value n bits left (n stored in SV register). Store the upper 16 bits of the 32-bit shift result to PH.

4.4.4 Class 4 Instructions: Address Register and Memory Reference

Class 4 instructions operate on the indirect register, R*x*, that exists in the address unit (ADU). Even though the last three registers (R5–R7) are special (INDEX, PAGE, and STACK), class 4 instructions uniformly apply to all registers. Subclass 4a provides transfers to and from memory. In indirect mode, any one auxiliary register can serve as the address for loading and storing the contents of another.

Subclass 4b instructions provide some basic arithmetic operations between referenced auxiliary register and short 8-bit constants from program memory. These instructions are included to provide efficient single cycle instructions for loop control and for software addressing routines.

Subclass 4c provide basic arithmetic operations between the referenced auxiliary register and 16-bit constants from program memory. These instruction require 2 instruction cycles to execute.

Also a compare to R5 (INDEX) is provided for efficient loop control where the final loop counter value is not chosen to be zero.

Table 4–21. Class 4a Instruction Encoding

Bit	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Class 4a	1	1	1	1	0	C4a	*R*			*adrs*							
Class 4b	1	0	1	1	C4b		*k4*	*k3*	*k2*	*k7*	*k6*	*k5*	*R*			*k1*	*k0*
Class 4c	1	1	1	1	1	1	1	0	0	0	C4c		*R*			x	x
Class 4d	1	1	1	1	1	1	1	0	0	1	C4d		*R*			x	x

Table 4–22. Class 4a Instruction Description

C4a	Mnemonic	Description
0	MOV {*adrs*}, R*x*	Store R*x* register to data memory referred by addressing mode {adrs}. Modify transfer status.
1	MOV R*x*, {*adrs*}	Load R*x* with the value in data memory referred by addressing mode {adrs}. Modify transfer status.

Table 4–23. Class 4b Instruction Description

C4b		Mnemonic	Description
0	0	ADDB R*x*, *imm8*	Add 8-bit positive constant to R*x* register. Modify RX status.
0	1	SUBB R*x*, *imm8*	Subtract 8-bit positive constant from R*x* register. Modify RX status.
1	0	MOVB R*x*, *imm8*	Load R*x* with the an 8-bit positive constant. Modify RX status.
1	1	CMPB R*x*, *imm8*	Store the status of the subtraction (R*x* – 8-bit positive constant) into RZF and RCF bits of the STAT register. R*x* remains unchanged.

Table 4–24. Class 4c Instruction Description

C4c		Mnemonic	Description
0	0	ADD R*x*, *imm16*	Add 16-bit positive constant to R*x* register. Modify RX status.
0	1	SUB R*x*, *imm16*	Subtract 16-bit positive constant from R*x* register. Modify RX status.
1	0	MOV R*x*, *imm16*	Load R*x* with the an 16-bit positive constant. Modify RX status.
1	1	CMP R*x*, *imm16*	Store the status of the subtraction (R*x* – 16-bit positive constant) into RZF and RCF bits of the STAT register. R*x* remains unchanged.

Table 4–25. Class 4d Instruction Description

C4d		Mnemonic	Description
0	0	ADD R*x*, R5	Add R5 to R*x* register, Modify RX status.
0	1	SUB R*x*, R5	Subtract R5 from R*x* register. Modify RX status.
1	0	MOV R*x*, R5	Load R*x* with R5. Modify RX status.
1	1	CMP R*x*, R5	Store the status of the subtraction (R*x* – R5) into RZF and RCF bits of the STAT register. R*x* and R5 remain unchanged.

4.4.5 Class 5 Instructions: Memory Reference

Class 5 instructions provide transfer to and from data memory and all registers except accumulators and R*x* which are included in classes 1 and 4. The registers referenced for both read and write operations are the multiplier register (MR), the product high register (PH), the shift value register (SV), the status register (STAT), the top of stack (TOS), the string register (STR), and the four accumulator pointer registers AP0 to AP3. The data pointer register (DP) is read only since its value is established by lookup table instructions. The RPT *n* (repeat) instruction is write only since repeated instructions cannot be interrupted. IRET and RET instructions are read only operations for popping the stack and are included in this class because the stack is memory mapped. Also included in this class are four flag instructions that modify flag memory and two instructions that multiply memory by MR, storing the results in the PH register.

Table 4–26. Class 5 Instruction Encoding

Bit	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Class 5	1	1	0	1	C5					*adrs*							
RET	1	1	0	1	1	1	1	0	0	0	1	1	1	1	1	1	0
IRET	1	1	0	1	1	1	1	0	1	0	1	1	1	1	1	1	0

Table 4–27. Class 5 Instruction Description

C5					Mnemonic	Description
0	0	0	0	0	MOV {*adrs*}, SV	Store SV in the data memory location referred by addressing mode {*adrs*}, zero filled on upper 12 bits. Transfer status is modified.
0	0	0	0	1	MOV {*adrs*}, PH	Store the PH in the data memory location referred by addressing mode {*adrs*}. Transfer status is modified.
0	0	0	1	0	MOV {*adrs*}, STAT	Store the status (STAT) register contents to the data memory location referred by addressing mode {*adrs*} (17 bits including TAG). No modification of status.
0	0	0	1	1	MOV {*adrs*}, STR	Store string (STR) register contents to data memory location referred by addressing mode {*adrs*}, zero filled on upper 8 bits. Transfer status is modified.
0	0	1	*n*		MOV {*adrs*}, AP*n*	Store the accumulator pointer (APn) register to the data memory location in {*adrs*}. The upper 10 bits are zero filled. Transfer status is modified.
0	1	0	0	0	MOV {*adrs*}, MR	Store the contents of the multiplier (MR) register in {*adrs*}. Transfer status is modified.
0	1	0	0	1	Reserved	
0	1	0	1	0	MOV {*adrs*}, DP	Store the data pointer (DP) register contents to the location referred by {*adrs*}. Transfer status is modified.

Table 4–27. Class 5 Instruction Description (Continued)

C5					Mnemonic	Description
0	1	0	1	1	MOV {*adrs*}, TOS	Store the contents of the top of stack (TOS) register to the data memory location referred by addressing mode {*adrs*}. Transfer status is modified.
0	1	1	0	0	STAG {*adrs*}	Store 1 to the 17th bit of data memory location referred by {*adrs*}. Set the tag bit.
0	1	1	0	1	RTAG {*adrs*}	Store 0 to the 17th bit of data memory location referred by {*adrs*}. Clear the tag bit.
0	1	1	1	*n*–1	MOVT {*adrs*}, TF*n*	Store TF1 bit if *n*=1, TF2 bit if *n*=0 status bit to 17th bit of data memory location referred by addressing mode {*adrs*}.
1	0	0	0	0	MOV SV, {*adrs*}$_4$	Load shift value (SV) register with contents of the location referred by addressing mode {*adrs*}. Transfer status is modified.
1	0	0	0	1	MOV PH, {*adrs*}	Load Product High (PH) register with content of data memory location value referred by addressing mode {*adrs*}. Transfer is status modified.
1	0	0	1	0	MOV TOS, {*adrs*}	Load top of stack (TOS) register with content of data memory location referred by addressing mode {*adrs*}.
1	0	0	1	1	MOV STR, {*adrs*}$_8$	Load String (STR) register with content of data memory location referred by addressing mode {*adrs*}. Only the lower 8 bits are loaded. Transfer status modified.
1	0	1	n	n	MOV AP*n*, {*adrs*}	Load lower 5 bits with content of data memory location referred by addressing mode {*adrs*} to accumulator pointer (AP) register *n*. Transfer status is modified (16-bit value).
1	1	0	0	0	MOV MR, {*adrs*}	Load Multiplier (MR) register with content of data memory location referred by addressing mode {*adrs*} and set the multiplier signed mode (UM=0 in STAT register). Transfer status is modified.
1	1	0	0	1	MOVU MR, {*adrs*}	Load Multiplier (MR) register with content of data memory location referred by addressing mode {*adrs*} and set the multiplier unsigned mode (UM=1 in STAT register). Transfer status is modified.
1	1	0	1	0	MULR {*adrs*}	Multiply MR register by content of data memory location referred by addressing mode {*adrs*}, add 0x00008000 to the 32-bit product to produce a rounding on the upper 16 bits. Store the upper rounded 16 bits to the PH register. No status change.
1	1	0	1	1	MUL {*adrs*}	Multiply MR register by content of data memory location referred by addressing mode {*adrs*} and store the most significant 16 bits of product into the PH register. No status change.
1	1	1	0	0	RET†	Return from subroutine. Load data memory location value addressed by R7 (STACK) to program counter.
1	1	1	0	1	IRET†	Return from interrupt routine. Load data memory location value addressed by R7 (STACK) to program counter.

† The entire 17 bit is encoded. See Table 4–26.

Table 4–27. Class 5 Instruction Description (Continued)

C5					Mnemonic	Description
1	1	1	1	0	RPT $\{adrs\}_8$	Load repeat counter with lower 8 bits of data memory location referred by addressing mode {*adrs*}. Interrupts are queued during execution.
1	1	1	1	1	MOV STAT, {*adrs*}	Load status (STAT) register with effective data memory location referred by addressing mode {*adrs*} (17 bits with TAG).

4.4.6 Class 6 Instructions: Port and Memory Reference

These instructions provide the basic expansion port of the MSP50P614/ MSP50C614 processor. IN instructions transfer 16-bit data from one of 16 expansion ports. OUT instructions transfer 16-bit data to one of the 16 expansion ports. In a typical system, the expansion ports are divided into those that serve internal peripheral functions and those that serve external pins. For subclass 6b, IN and OUT provide bidirectional transfers between the same port address (16) and accumulator. In addition, IN and OUT instructions in class 6b can communicate with an extra 48 ports (a total of 64 including the shared ports). Class 6b instructions also have reference to the string bit for checking the arithmetic status of a string transfer.

Table 4–28. Class 6a Instruction Encoding

Bit	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Class 6a	1	1	0	0	C6a	*port4*				*adrs*							
Class 6b	1	1	1	0	1	1	s	A*n*		*port6*						C6b	~A~

Table 4–29. Class 6a Instruction Description

C6a	Mnemonic	Description
0	IN {*adrs*}, *port4*	Transfer a 16-bit value of addressed port to data memory location referred by addressing mode {*adrs*}. Refer to port address map. Transfer status is modified.
1	OUT *port4*, {*adrs*}	Transfer a 16-bit value in the data memory location referred by addressing mode {*adrs*} to addressed port. Refer to Port address map. Transfer is status modified.

Table 4–30. Class 6b Instruction Description

C6b	Mnemonic	Description
0	IN A*n*[~], *port6* INS A*n*[~], *port6*	Transfer the port's 16-bit value to an accumulator. Port addresses 0–63 are valid. ALU status is modified.
1	OUT *port6*, A*n*[~] OUTS *port6*, A*n*[~]	Transfer a 16-bit accumulator value to the addressed port. Port addresses 0–63 are valid. Transfer status is modified.

4.4.7 Class 7 Instructions: Program Control

This class of instructions provides the logical program control of conditional branches (jumps) and calls (subroutines).

Both branch and call instructions require a 32-bit instruction word. The first word contains the opcode and condition fields and the second word contains the destination address. The condition field can specify the true (*Not*=0) or false (*Not*=1) condition of 22 different status conditions. The status bits that establish the conditions are latched and remain unchanged until another instruction that affects them is executed.

In addition to call, a macro-call instruction is included. This instruction is similar to an unconditional call instruction. When executed it pushes the PC+1 value to the STACK and loads a paged vector (7F loaded in the upper 8 bits of PC and an 8-bit vector number loaded into the lower 8 bits of the PC). This makes the macro–call a single word instruction that take 2 instruction cycles to execute. This instruction is useful for referencing frequently used subroutines. A normal RET instruction is used to return to the main program from macro-calls.

Auxiliary register R7 (STACK) is used as the program stack pointer and is automatically incremented on calls and macro-calls. It is automatically decremented on returns. Interrupts are vectored in the same way as macro-calls. The stack pointer is incremented when interrupts fire and decremented when an IRET is executed. One side effect of the program stack's operation is that it is not permissible to return to a RET instruction. Either the compiler inserts a NOP between such occurrences or the programmer must avoid this sequence.

Table 4–31. Class 7 Instruction Encoding and Description

Bit	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VCALL *vector8*	1	1	1	1	1	1	1	0	1	*vector8*							
J*cc*	1	0	0	0	0	0	Not	*cc*					R*x*			*pm*	
JMP *A*n*	1	0	0	0	1	0	x	A*n*		x							
C*cc*	1	0	0	0	0	1	Not	*cc*					x				
CALL *An	1	0	0	0	1	1	x	A*n*		x							

cc					*cc names*		**Description**
					cc name	*Not cc name*	
0	0	0	0	0	Z	NZ	Conditional on ZF=1
0	0	0	0	1	S	NS	Conditional on SF=1
0	0	0	1	0	C	NC	Conditional on CF=1
0	0	0	1	1	B	NB	Conditional on ZF=0 and CF=0
0	0	1	0	0	A	NA	Conditional on ZF=0 and CF=1
0	0	1	0	1	G	NG	Conditional on SF=0 and ZF=0
0	0	1	1	0	E	NE	Conditional if ZF=1 and OF=0
0	0	1	1	1	O	NO	Conditional if OF=1
0	1	0	0	0	RC	RNC	Conditional on RCF=1
0	1	0	0	1	RA	RNA	Conditional on RZF=0 and RCF=1
0	1	0	1	0	RE	RNE	Conditional on RZF=1
0	1	0	1	1	REZI		Conditional on value of R*x*=0 (Not available on Calls)
0	1	1	0	0	RLZI		Conditional on MSB of R*x*=1. Not available on Calls.
0	1	1	0	1	L	NL	Conditional on ZF=0 and SF=1
0	1	1	1	0			Not assigned
0	1	1	1	1			Not assigned
1	0	0	0	0	TF1	NTF1	Conditional on TF1
1	0	0	0	1	TF2	NTF2	Conditional on TF2
1	0	0	1	0	TAG	NTAG	Conditional on TAG
1	0	0	1	1	IN1	NIN1	Conditional on IN1 status
1	0	1	0	0	IN2	NIN2	Conditional on IN2 status

Table 4–31. Class 7 Instruction Encoding and Description (Continued)

cc					*cc names*		**Description**
					cc name	*Not cc name*	
1	0	1	0	1			Unconditional
1	0	1	1	0			Not assigned
1	0	1	1	1			Not assigned
1	1	0	0	0	XZ	XNZ	Conditional on XSF
1	1	0	0	1	XS	XNS	Conditional on XZF
1	1	0	1	0	XG	XNG	Conditional on ! XSF and ! XZF
1	1	0	1	1			Not assigned
1	1	1	0	0			Not assigned
1	1	1	0	1			Not assigned
1	1	1	1	0			Not assigned
1	1	1	1	1			Not assigned

4.4.8 Class 8 Instructions: Logic and Bit

This class of instructions provides a flexible and efficient means to make complex logical decisions. Instead of making a sequence of single bit decisions and constructing a logical statement through a branch decision tree, the program can sequentially combine several status conditions to directly construct a final logic value (TF1 or TF2) which can be used to control a subsequent branch or call. This class includes two subclasses. Class 8a instructions update one of the test flags (TF1 or TF2) with a logical combination of the old test flag value and an addressed memory flag value. Subclass 8b provides a flexible means of logically combining the test flag (TF1 or TF2) with a status condition and storing the results back to the test flag.

Table 4–32. Class 8a Instruction Encoding

Bit	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Class 8a	1	0	0	1	1	*flg*	*Not*	C8a			*flagadrs*						
Class 8b	1	0	0	1	0	*flg*	*Not*	*cc*					Rx			C8b	
RFLAG {*flagadrs*}	1	0	0	1	1	0	0	0	1	1	*flagadrs*						
SFLAG {*flagadrs*}	1	0	0	1	1	1	0	1	0	1	*flagadrs*						

Table 4–33. Class 8a Instruction Description

C8a			Mnemonic	Description
0	0	0	MOV TF*n*, {*flagadrs*}	Load flag bit (17th bit) from data memory referred by flag addressing mode {*flagadrs*} to either TF1 or TF2 in status register. Load with inverted value if *Not*=1.
0	1	0	OR TF*n*, {*flagadrs*}	Logically OR either TF1 or TF2 with flag bit (17th bit) from data memory referred by flag addressing mode {*flagadrs*} (or inverted value if N=1) addressed by the instruction and store back to TF1 or TF2 respectively.
1	0	0	AND TF*n*, {*flagadrs*}	Logically AND either TF1 or TF2 with flag bit (17th bit) from data memory referred by flag addressing mode {*flagadrs*} (or inverted value if *Not*=1) addressed by the instruction and store back to TF1 or TF2 respectively.
1	1	0	XOR TF*n*, {*flagadrs*}	Logically exclusive OR either TF1 or TF2 with flag bit (17th bit) from data memory in {*flagadrs*} if *Not*=1(or inverted value if *Not*=0) addressed by the instruction and store back to TF1 or TF2 respectively.
0	0	1	MOV {*flagadrs*}, TF*n*	Store TF1 or TF2 to flag bit (17th bit) from data memory referred by flag addressing mode {*flagadrs*}.
Table 4–32			RFLAG {*flagadrs*}	Reset flag bit (17th bit) from data memory referred by flag addressing mode {*flagadrs*}.to 0
Table 4–32			SFLAG {*flagadrs*}	Set flag bit (17th bit) from data memory referred by flag addressing mode {*flagadrs*}.to 1

Table 4–34. Class 8b Instruction Description

C8b		Mnemonic	Description
0	0	MOV TF*n*, {*cc*} [, R*x*]	Load a logic value of the tested condition to one of the test flag bits in status register (TF1 or TF2).
0	1	OR TF*n*, {*cc*} [, R*x*]	Logically modify one of the two test flags in status register (TF1 or TF2) by ORing it with the status condition specified.
1	0	AND TF*n*, {*cc*} [, R*x*]	Logically modify one of the two test flags in status register (TF1 or TF2) by ANDing it with the status condition specified.
1	1	XOR TF*n*, {*cc*} [, R*x*]	Logically modify one of the two test flags in status register (TF1 or TF2) by EXCLUSIVE ORing it with the status condition specified. For this instruction the polarity of *Not* is inverted (*Not*=1 for XOR, Not=0 for XNOR).

4.4.9 Class 9 Instructions: Miscellaneous

This instruction class includes all the remaining instructions that do not fit in the previous classes. Some instructions have byte wide operand fields and others have no operands. One subclass is a set of instructions that provide specific DSP functions (FIR filters). Another subclass provides some hardware/ software loop capability. Ten instructions provide the means to set or reset five different status mode bits independently.

Table 4–35. Class 9a Instruction Encoding

Bit	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Class 9a	1	1	1	0	1	0	0	A*n*		C9a		0	R*x*			1	1
Class 9b	1	1	1	1	1	1	0	C9a		*imm8*							
Class 9c	1	1	1	1	1	0	1	AP*n*		0	C9c	x	*imm5*				
Class 9d	1	1	1	1	1	1	1	1	0	C9d				0	0	0	0
ENDLOOP *n*	1	1	1	1	1	1	1	1	0	0	0	0	1	0	0	0	n
NOP	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Table 4–36. Class 9a Instruction Description

C9a		Mnemonic	Description
0	0	FIRK A*n*, *R*x*	Finite impulse response tap execution. When used with repeat counter will execute a 16 bit ×16 bit multiplication between an indirect-addressed data memory buffer and program memory (coefficients). 32-bit accumulation. Circular buffering. Each tap executes in 2 cycles. R*x* automatically increments by 2 per tap.
0	1	FIR A*n*, *R*x*	Finite impulse response tap execution. When used with the repeat counter, it will execute a 16 bit ×16 bit multiplication between two indirect-addressed data memory buffers into a 32-bit accumulator. Circular buffer operation. Executes in 2 instruction cycles. R*x* and R(*x*+1) automatically increments by 2 per tap.
1	0	CORK A*n*, *R*x*	Correlation function. When used with repeat will execute 16×16 multiplication between data memory and program memory, 48-bit accumulation, and a circular buffer operation. Each tap takes 3 instruction cycles. R*x* automatically increments by 2 per tap.
1	1	COR A*n*, *R*x*	Correlation function. When used with repeat will execute 16×16 multiplication between two indirectly addressed data memory buffers, 48-bit accumulation, and a circular buffer operation. Each tap takes 3 instruction cycles. R*x* and R(*x*+1) automatically increments by 2 per tap.

Table 4–37. Class 9b Instruction Description

C9b		Mnemonic	Description
0	0	RPT *imm8*	Load the repeat counter with an 8-bit constant and execute the instruction that follows *imm8*+2 times. Interrupts are queued during execution.
0	1	MOV STR, *imm8*	Load the **STR** register with an 8-bit constant.
1	0	MOV SV, *imm4*	Load the **SV** (shift value) register with a 4-bit constant.

Table 4–38. Class 9c Instruction Description

C9c	Mnemonic	Description
0	MOV AP*n*, *imm6*	Load the accumulator pointer (**AP**) with a 5-bit constant.
1	ADD AP*n*, *imm5*	Add a 5-bit constant *imm5* to the referenced accumulator pointer(**AP**).

Table 4–39. Class 9d Instruction Description

C9d				Mnemonic	Description
0	0	0	0	BEGLOOP	Marks the beginning of loop. Queue interrupts and pushes the next PC value onto a temporary stack location.
0	0	0	1	ENDLOOP *n*	If R4 is not negative, pops the temporary stack value back on the PC and decrements R4 by *n*. If R4 is negative, the instruction is a NOP and execution will exit the loop. *n* is either 1 or 2
0	0	1	0	IDLE	Stops processor clocks. Device enters low power mode waiting on an interrupt to restart the clocks and execution.
1	0	0	0	INTE	Sets IM bit in status register to a 1, thus enabling interrupts.
1	0	0	1	INTD	Sets IM bit in status register to a 0, thus disabling interrupts.
1	0	1	0	SXM	Sets XM in status register to 1 enabling sign extension mode.
1	0	1	1	RXM	Sets XM in status register to 0, disabling sign extension mode.
1	1	0	0	SFM	Sets FM in status register to 1, enabling multiplier shift mode for signed fractional arithmetic.
1	1	0	1	RFM	Sets FM in status register to 0, enabling multiplier shift mode for unsigned fractional or integer arithmetic.
1	1	1	0	SOVM	Set OM bit in status register to 1, enabling ALU saturation output (DSP mode).
1	1	1	1	ROVM	Set OM bit in status register to 0, disabling the saturating ALU operation (normal mode).

4.5 Bit, Byte, Word and String Addressing

The MSP50P614/MSP50C614 has instructions which address bits, bytes, words and strings in data memory or program memory. Data memory is always accessed in bytes by the hardware, but is based on the instruction. The data memory location is treated as a byte, word, or flag address. There are five different kinds of addresses: byte addresses, byte-string addresses, word addresses, word-string addresses, and flag addresses. Each type of address is described below. Refer to Figure 4–3 and Table 4–40 for reference.

Byte and byte string address: Byte addressing is used to access individual bytes with an instruction in byte mode. Such instructions have a suffix, B, at the end of instruction name (for example, ADDB, MOVB, etc.). A byte string

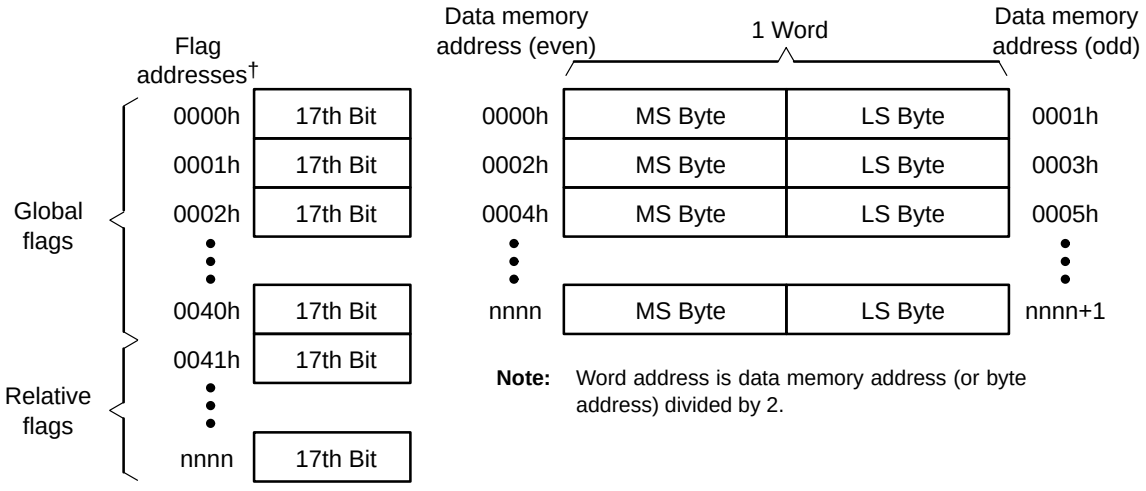
is a string of bytes. The length of the byte string is stored in the string register (STR). To define the length of a string, the STR register should hold the length of the string minus 2. For example, if the length of a byte string is 10, then STR should be 8. A byte string address can be even or odd. Byte string data is fetched from the lower address (starting address) one byte at a time to consecutive addresses.

NOTE: Data Memory Access

Data memory access (RAM) is always accessed with byte addresses. Program memory (ROM) is accessed with 17-bit words. R*x* registers autoincrement (or autodecrement) by 1 for byte addressing, by 2 for word addressing, or by the length of the string in bytes if R*x*++ (or R*x*– –) is used.

Word and Word String Addresses: One data memory word is composed of two consecutive bytes. A word address is always an even byte address and the least significant bit of the byte address is assumed to be zero. Instructions that operate on words have internal hardware which increments the byte address appropriately to load the two consecutive bytes in one clock cycle. To use an absolute word address, the address should be multiplied by 2. A word-string is a string of consecutive words. Like a byte-string, word-strings use the STR register to define the string length. Word-strings always start at an even byte address. When string instructions are used, words are fetched from the first word-string memory location to consecutive addresses. The word address is the data memory address in bytes. This is obtained by multiplying the byte address by two.

Figure 4–3. Data Memory Organization and Addressing



† Flag address always accesses the 17th bit of 17 bit wide data word in data memory.

Flag Address: The flag (or TAG) address uses linear addressing from 0 to the size of data memory in 17-bit wide words (0 to 639 for MSP50P614/ MSP50C614). Only the 17th bit is accessible. When a word memory location is read, the corresponding flag for that location is always loaded into the TAG bit of the status register (STAT). The flag address always corresponds to a 17-bit wide word address. If string instructions are used, then the flag bit of the last memory location of the string is loaded into the TAG bit of the status register. Global flag addressing or relative flag addressing is used to address flags. Flag bits can be set or reset using flag instructions in addition to various logical operations. The flag address does not have a string mode.

Rx Post Modifications: Indirect addressing allows post modification of R*x*. For byte and byte-string mode, R*x* is post modified by 1 for each byte. For word and word-string mode R*x* is post modified by 2 for each word. Post modification of R*x* is not available for flag addressing.

Table 4–40. Data Memory Address and Data Relationship

Mode	Address Used	Data Order	Rx Post modify†
Single byte	Absolute 16-bit address	8-bit data	1
Byte string	Beginning of string at lower address	String length times 8-bit data by Incrementing addresses	1 per byte in string
Single word	Even address, if odd address is used, the LSB bit of address is assumed 0	16-bit data	2
Word string	Even address beginning at a lower address; if odd address is used, the LSB bit of address is assumed 0	String length times 16-bit data by incrementing addresses	2 per word in string
Flag	Address is considered as holding 17-bit data, but only 17th bit is accessed.	1-bit data	not available

† R*x* post modification is available by various addressing modes (see 4.3, Instruction Syntax and Addressing Modes for detail).

Example 4.5.1
```
MOVB A0, *0x0003
```
Refer to Figure 4–4 for this example. This instruction loads the value 0x78 to the accumulator. The upper 8 bits of the accumulator is padded with zeros.

Example 4.5.2
```
MOV A0, *0x0000
MOV A0, *0x0001
```
Refer to Figure 4–4 for this example. Both instructions will load the value 0x1234 to the accumulator. In word addressing, the LSB bit of the address is assumed to be zero. Thus, in the second instruction, the least significant bit of the address is ignored.

Example 4.5.3
```
MOV A0, *0x0004 * 2
```
Refer to Figure 4–4 for this example. The word address 0x0004 is referred. Multiplication by 2 is necessary to convert the word address into the equivalent byte address . After multiplication, the byte address is 0x0008. This instruction will load the value 0x1122 to the accumulator.

Figure 4–4. Data Memory Example

Absolute Word Memory Location	Data Memory Location (even) = 2 * (Absolute word memory location)	MS Byte	LS Byte	Data Memory Location (odd)
0x0000	0x0000	0x12	0x34	0x0001
0x0001	0x0002	0x56	0x78	0x0003
0x0002	0x0004	0x9a	0xbc	0x0005
0x0003	0x0006	0xde	0xf0	0x0007
0x0004	0x0008	0x11	0x22	0x0009
0x0005	0x000a	0x33	0x44	0x000b

Example 4.5.4

```
MOV STR, 4–2
MOV AP0, 2
MOVBS A0, *0x0003
```

Refer to Figure 4–4 for this example. The byte-string length is 4. It is loaded to the string register (STR) in the first instruction. AP0 is 2 and it points to AC2. Third instruction loads the value of the string at byte address, 0x0003, and subsequently stores its contents into four consecutive accumulators starting from AC2. The result is, AC2 = 0x0078, AC3 = 0x009A, AC4 = 0x00BC, AC5 = 0x00DE.

Example 4.5.5

```
MOV STR, 4–2
MOV AP0, 2
MOVS A0, *0x0003
```

Refer to Figure 4–4 for this example. The byte-string length is 4. AP0 is loaded with 2 and points to AC2. The third instruction loads the value of the string at address 0x0002 (LSB bit is assumed 0) and stored into four consecutive accumulators starting from AC2. The result is, AC2 = 0x5678, AC3 = 0x9ABC, AC4 = 0xDEF0, AC5 = 0x1122. Same result can be obtained by replacing the third instruction by,

```
MOVS A0, *0x0001 * 2
```

which uses the absolute word memory address.

Example 4.5.6

```
MOV STR, 4–2
OV AP0, 2
MOV R0, 0x0005
MOVBS A0, *R0++
```

Refer to Figure 4–4 for this example. The byte string length is 4. AP0 points to AC2. R0 is loaded with 0x0005. The fourth instruction loads the value of the byte-string at the address in R0 (i.e, 0x0005 in byte mode). R0 auto–increments by 1 after every fetch and stores the RAM contents into four consecutive accumulators starting from AC2. The result is, AC2 = 0x00BC, AC3 = 0x00DE, AC4 = 0x00F0, AC5 = 0x0011. There were four byte fetches and the new value of R0 = 0x0009.

Example 4.5.7

```
MOV STR, 4–2
MOV AP0, 2
MOV R0, 0x0001 * 2
MOVBS A0, *R0++
```

Refer to Figure 4–4 for this example. The word-string length is 4. AP0 points to AC2 accumulator. R0 is loaded with 0x0002. The fourth instruction loads the value of the word-string at the RAM address in R0, 0x0002. R0 autoincrements by 2 after each fetch and stores them into four consecutive accumulators starting from AC2. The result is, AC2 = 0x5678, AC3 = 0x9ABC, AC4 = 0xDEF0, AC5 = 0x1122. There were 4 word fetches and the new value of R0 = 0x000A.

Example 4.5.8

```
SFLAG *0x0003
MOV A0, *0x0003 * 2
RFLAG *0x0003
MOV A0, *0x0003 * 2
```

Refer to Figure 4–4 for this example. This example illustrates the use of the TAG and flag bits. Notice that SFLAG uses a word address, 0x0003, while the MOV instruction uses a byte address 0x0003 * 2. The first instruction sets the flag/tag bit at flag address 0x0003. Flag address 0x0003 represents the 17th bit of the 3rd word (or 6th byte) of RAM. In the second instruction, this flag bit is placed in the TAG status bit of the STAT and the value in RAM location 0x0003 * 2 is placed in A0. The third instruction resets the flag/tag to 0 at the same flag address. The fourth instruction reads the same word memory location and writes the TAG bit of STAT, which is now 0. Note: SFLAG *0x0003 could have been replaced by STAG *0x0003 * 2 and RFLAG *0x0003 could have been replaced by RTAG *0x0003 * 2.

Example 4.5.9

```
SFLAG *0x0005
MOVB A0, *0x000b
RFLAG *0x0005
MOVB A0, *0x000b
```

Refer to Figure 4–4 for this example. The SFLAG instruction sets the 17th bit (tag/flag) of the 5th word of RAM. The MOVB instruction gets the lower byte of the 5th word of RAM and puts it in A0. In addition, the TAG bit of the STAT register is set. If the MOVB instruction addressed *0x000A instead of *0x000B, the STAT register would still be updated with the same tag/flag bit (the 17th bit of the 5th word of RAM). This means that odd byte locations in RAM, RAM_{odd}, have the same tag/flag as the preceding byte location RAM_{odd} –1. For example, the 7th word of RAM is made up of two bytes: 0x000E, and 0x000F. These two byte locations share the same tag/flag bit.

Example 4.5.10

```
MOV STR, 0
SFLAG *0x00032
MOVS A0, *0x0031 * 2
RFLAG *0x00032
MOVS A0, *0x0031 * 2
```

Refer to Figure 4–4 for this example. This example is to illustrate the effect of the tag/flag bit when used with a string instruction. The string register (STR) is loaded with 0 (string length of 2). The second instruction sets the flag bit to 1 at flag address 0x0032. The next instruction reads the word-string at word memory location, 0x0031, into A0 and also sets the TAG bit of STAT to 1 corresponding to the last memory location of the string (which is word address 0x0032 in this case). The next two instructions verify this by setting the flag to zero and reading the memory string again.

4.6 MSP50P614/MSP50C614 Computational Modes

MSP50P614/MSP50C614 has the following computational modes which are the first 4 bits of the status register.

- ❑ Sign extension mode (bit 0 or XM bit of STAT)
- ❑ Unsigned mode (bit 1 or UM bit of STAT)
- ❑ Overflow mode (bit 2 or OM bit of STAT)
- ❑ Fractional mode (bit 3 or FM bit of STAT)

These modes can be set by setting the appropriate status register bits or by special instructions (Class 9) as shown in Table 4–41.

Table 4–41. MSP50P614/MSP50C614 Computational Modes

Computational Mode	Setting Instruction	Resetting Instruction	Function
Sign extension	SXM	RXM	STAT.XM = 1 produces sign extension on data as it is passed into accumulators. This mode copies the 16th bit of the data in the multiplier/multiplicand to the 17th bit. This causes signed multiplication of two signed numbers. STAT.XM = 0 suppresses sign extension.
Unsigned	none	none	STAT.UM = 1 causes unsigned multiplication where the multiplier assumes its arguments as unsigned value. MOVU instruction can be used to enable this mode. STAT.UM = 0 disables unsigned multiplication.
Overflow	SOVM	ROVM	STAT.OM = 1 initiates overflow mode. Overflows cause the accumulator to acquired the most positive or most negative value. In the case of string values, only the MSB 16 bits are modified. The remaining bits in the string are unchanged. STAT.OM = 0 normal overflow operation and the accumulator content is unchanged if any overflow occurs. Affects OF bit of STAT in case of overflow.
Fractional	SFM	RFM	STAT.FM = 1 enables fractional multiplication shift mode. The multiplier is shifted left 1 bit to produce a 17 bit operand. This mode is used on signed binary fractions and does not require the user to left shift as it would have been required if the FM bit was not set. STAT.FM = 1 turns off fractional mode.

Sign Extension Mode: Sign extension mode can be enabled/disabled by setting/resetting the XM bit of STAT. When in sign extension mode, a multiply operation will copy the 16th bit of the multiplier/multiplicand to the 17th bit. When multiplied, this will give a 17 x 17 bit multiplication producing 34-bit result where the upper two bits (33rd and 34th bits) are the sign bits and discarded by the processor. Sign extension is also applicable in string mode. Sign extension mode is the recommended mode to use for signed number multiplication.

Example 4.6.1

```
SXM
MOV A0, 0x8000
MOV MR, 0x8000
MULTPL A0, A0
```

This example illustrates the sign extension mode during multiplication. Here, two negative number 0x8000 are multiplied with 0x8000 to obtain a positive number 0x40000000. If the signs were not extended, we would have obtained 0xC0000000, a negative number.

Example 4.6.2

```
SXM
MOV STR, 2–2    ; string length=2
MOV MR, 0x8000
MOV A0, 0x8000, ++A ; load MS Byte
MOV A0, 0x0000, ––A ; load LS Byte
MULTPLS A0, A0
```

This example illustrates the sign extension mode on a string during multiplication. Here, two negative numbers 0x80000000 and 0x8000 are multiplied to obtain a positive number 0x400000000000. If the signs were not extended, we would have obtained 0xC00000000000, a negative number.

Unsigned Mode: The multiplier unsigned mode may be enabled/disabled by setting/resetting the UM bit of the STAT. When in unsigned mode, the 17^{th} bit of the multiplier is loaded as zero to indicate an unsigned value. When UM is set to zero, signed multiplication is enabled and the multiplier copies the MSB of the multiplier (16^{th} bit) to the 17^{th} bit of the multiplier.

Example 4.6.1

```
MOV A0, 0x8000
MOVU MR, A0
MOV A0, 0x80
MULTPL A0, A0
```

In this example, we do an unsigned multiplication between 0x8000 and 0x80. The first two lines set up the MR register with value 0x8000 and switch to unsigned multiplication mode. Line 3 loads A0 with 0x80 and line 4 multiplies the values in unsigned mode. The lower 16 bits of the result is stored in A0 and the upper 16 bits are stored in PH. The final result is 0x400000, where PH holds the value 0x0040 and A0 holds the lower 16 bits. Notice that if the multiplication is not done in unsigned mode, the MR is treated as negative. We would have obtained 0xFFC00000 (PH = 0xFFC0, A0 = 0000), which is the negative value of the previous result. The key to unsigned multiplication is the MOVU instruction in the second line which set the UM bit to 1 in the STAT register and switches the multiplication mode to unsigned.

Overflow Mode: The accumulator's overflow mode may be enabled/disabled by setting/resetting the OM bit of STAT. When the computation is in the overflow mode and an overflow occurs, the overflow flag is set and the accumulator is loaded with either the most positive or the most negative value representable in the accumulator, depending upon the direction of the overflow. In string mode, instead of representing the most positive or most negative value, only the 16-bit MSB is set to 0x7FFF or 0x8000 depending on direction of overflow. The remaining words of the accumulator string are unchanged. If the OM status register bit is reset and an overflow occurs, the overflowed results are placed in the accumulator without modification. Note that logical operations cannot result in overflow.

Example 4.6.1

```
SOVM
MOV A0, 0x7FFE
ADD A0, 5
```

In this example, we set the overflow mode (OM = 1 of STAT). Adding 0x7FFE with 5 causes an overflow (OF = 1 of STAT). Since the expected result is a positive value, the accumulator saturates to the largest representable value, 0x7FFF. If overflow mode was not set before the ADD instruction, then the accumulator would overflow. Therefore, the result, 0x8003, would be a negative value.

Example 4.6.2

```
SOVM
MOV STR, 2-2    ;string length = 2
MOV AP0, 0
MOV A0, 0x1234
MOV A0~, 0x1000
MOV A0, 0x7F00, ++A
MOV A0~, 0x1000
MOV AP0,0       ;point to beginning
                ;of string
ADD A0, A0~, A0
```

In this example, saturation on a string value is illustrated. A 2 word string is loaded into the STR register. The accumulator string, A0, is loaded with 0x7F001234 and accumulator string A0~ is loaded with 0x10001000. When the two values are added together, it causes an overflow. The OF bit of the STAT is set to 1, the 16-bit MSBs of the string become 0x7FFF, and the lower bits of the string become 0x2234. The final result is 0x7FFF2234. Note that if overflow mode was not set, the result would have been 0x8F002234.

Fractional Mode: Multiplier fractional mode may be enabled/disabled by setting/resetting the FM bit of STAT. When the multiplier is in fractional mode, the multiplier is shifted left 1 bit to form a 17 significant bit operand. Fractional mode avoids a divide by 2 of the product when interpreting the input operands as signed binary fractions (Q formats). Fractional mode works with string mode as well.

Example 4.6.1

```
SXM
MOV   A0, 0x7FFF
MOV   MR, 0x7FFF
MULTPL A0, A0   ;0x7FFF * 0x7FFF
                ;PH = 0x3FFF A0~ = 0001
SFM
MULTPL A0~,A0   ;PH = 0x7FFE A0~ = 0002
```

This example illustrates the differences between a regular multiply and a fractional mode multiply. The first multiply in the above code is nonfractional. The

high word of the result is stored in the PH register and is 0x3FFF. The low word is stored in A0~ as 0x0001. If the two numbers are considered as Q15 fractional numbers (all bits are to the right of the decimal point), then the result will be a Q30 number. To translate a Q30 number back to a Q15 number, first left shift the number (MOV A0,PH, SHL A0,A0), and then truncate the lower word (ignore A0~). When fractional mode is set, the left shift is done automatically (MOV A0,PH). Thus, the desired Q15 result is already in the PH register.

4.7 Hardware Loop Instructions

These instructions enhance both execution speed and code space requirements for procedures that use short loop sequences. Because of pipeline delays and the software overhead associated with counting, comparing and branching, software controlled structures are very inefficient for short loops. To ease this burden, two basic types of hardware assisted loop structures are included in the MSP50P614/MSP50C614 processor. Hardware loop instructions are summarized in Table 4–42.

Repeatable Instructions: Most instructions can be repeated N+2 times with zero software overhead. Repeated instructions are functionally identical to coding the same instruction N+2 times in sequence. Repeat loops require a RPT instruction to set a count length, N. This immediately precedes the instruction to be repeated. This next instruction is repeated N+2 times. The RPT instruction is useful for clearing RAM locations, filtering, etc. If the repeating instruction utilizes auto–increments/decrements to either Rx or AC registers (i.e. *R2++ or ++A), then the repeated modification controls will be permanent. If the repeatable instruction is a string instruction, then the string register (STR) will be replaced by N. During the execution of a RPT instruction, interrupts are queued. Queued interrupts are serviced after the RPT operation completes according to their priority.

String Instructions: String loops are enabled by direct field decodes in classes 1, 2b, 3 and 6b and have no counter overhead. These instructions automatically load the counter using the contents of the STR. String instruction loops are different because they assume the references made to data memory and accumulators are long data strings, causing pointers to auto–increment. Incrementing pointers does not affect the permanent value stored in R*x* or AP*n* registers. For arithmetic string operations, carries from one word operation will automatically be linked to the carry in of the next word operation. Additionally, status *equal to zero* will be detected on the result as a long string. These combinations provide efficient and convenient means to operate between lists or stings or between a fixed location and a list or string. All string instructions have a suffix, S. In this text, string instructions are written as *name*S. During

the execution of *a* string instruction, interrupts are queued. Queued interrupts are serviced according to their priority after the string operation is complete.

In addition to repeat and string instructions, the combination of repeated string instructions has a very useful function. Since there is only one counter to control the hardware repeat count, it is not possible to nest repeats and strings. When a repeat instruction is followed by a string instruction the string register count is replaced by the value in the preceding repeat instruction. This offers greater utility in some programs and avoids load and store operations on the string register.

Loop Instructions: This is a software loop with an explicit reference to R4. The beginning of the loop is marked with the BEGLOOP instruction which pushes the next sequential address to a temporary register. A second instruction, ENDLOOP, marks the end of the loop. When executed, ENDLOOP loads the temporary register to the program counter if R4 is positive and then post decrements R4. If R4 is negative, the program counter executes a NOP instruction and exits the loop. Since interrupts are queued during the execution of the loop, no provision for saving the contents of the temporary register is made. Interrupts, if enabled before the execution of BEGLOOP, will automatically be re-enabled after exiting the loop. Enabling interrupts inside the loop have no effect. Queued interrupts are processed according to their priority after the loop exits provided the corresponding interrupt is enabled. The loop overhead is 1 instruction cycle per loop cycle, ideal for repeating high priority repeated blocks in DSP routines.

Table 4–42. Hardware Loops in MSP50P614/MSP50C614

Syntax	Operation	Limitations
RPT *imm8* | $\{adrs\}_8$ {repeatable instruction}	{repeatable instruction} is executed n_R+2 times, where n_R is the value in repeat counter. If the instruction following RPT is a string instructions, then string length used will be n_R, not the value in the STR register. All interrupts are queued during loop execution. Queued interrupts are processed according to priority after the completion of the RPT loop.	$0 \le n_R \le 255$
{STR= n_S} {string instruction}	String length for the {string instruction} is n_S+2. All interrupts are queued during loop execution. Queued interrupts are processed according to priority after the completion of the {string instruction}. The maximum accumulator string length is 32, i.e., $0 \le n_S \le 29$.	$0 \le n_S \le 255$ NOTE: $0 \le n_S \le 29$ for accumulator strings.
{R4= N_{LOOP}} BEGLOOP {...body of loop...} ENDLOOP	The number of times the {...body of loop...} is executed is N_{LOOP}+2. All interrupts are queued during loop execution. Queued interrupts are processed according to priority after the completion of the BEGLOOP/ENDLOOP block.	$0 \le N_{LOOP} \le 32767$

4.8 String Instructions

Class 1, 2, 3, and 6 instructions can have string modes. During the execution of string instruction, STR register value plus 2 is assumed as string length. An accumulator string is a group of consecutive accumulators spanning from A*n* to the next N consecutive accumulators (N is the length of the string). The STR register should be loaded with N–2 to define a string length, N. A value of zero in the STR register defines a string length of 2 (string length 1 means the instruction is not in string mode). Arithmetic string instructions treat the string as an N word arithmetic value. The result is also an arithmetic value of the same length. Conditionals are set as they would be set without string mode. Comparing two strings is equivalent to comparing each bit of the string. The accumulator status is modified representing the outcome of the entire operation. Examine the following examples.

Table 4–43. Initial Processor State for String Instructions

Registers (*register# = value*)			
AP0 = 2	AP1 = 21 (0x15)	AP2 = 11 (0x0B)	AP3 = 29 (0x1D)
AC0 =	AC1 =	AC2 =	AC3 =
AC4 =	AC5 =	AC6 =	AC7 =
AC8 =	AC9 =	AC10 =	AC11 = 0xAAAA
AC12 = 0xAAAA	AC13 = 0xAAAA	AC14 = 0xAAAA	AC15 = 0xAAAA
AC16 =	AC17 =	AC18 =	AC19 =
AC20 =	AC21 = 0x1223	AC22 = 0xFBCA	AC23 = 0x233E
data memory (**address = data*)			
*0x0200 = 0x12AC	*0x0201 = 0xEE34	*0x0202 = 0x9086	*0x0203 = 0xCDE5
program memory (**address = data*)			
*0x1400 = 0x0123	*0x1401 = 0x4567	*0x1402 = 0x89AB	*0x1403 = 0xCDEF
*0x1404 = 0xFEDC	*0x1405 = 0xBA98	*0x1404 = 0x7654	*0x1405 = 0x3210

Example 4.8.1

```
MOV STR, 4–2;  string length = 2
MOVS A0, 0x1400
```

Refer to initial the processor state in Table 4–43. A0 points to AC2. Consider a program memory location string of length 4 at 0x1400 = 0xCDEF89AB45670123. STR equal to 4–2=2, defines a string length of 4. Final result, AC2=0x0123, AC3=0x4567, AC4=0x89AB, and AC5=0xCDEF,

Example 4.8.2

```
MOV STR, 3–2;  string length = 3
ADDS A1~, A1, *0x0200
```

Refer to the initial processor state in Table 4–43. A1 is AC21, A1~ is AC5, the

A1 string is 0x233EFBCA1223 and *0x200 = 0x9086EE3412AC. STR = 3–2=1, defines a string length of 3. Final result, A1~ string = 0x233EFBCA1223 + 0x9086EE3412AC = 0xB3C5E9FE24CF, AC5=0x24CF, AC6=0xE9FE, AC7=0xB3C5, STR=2 (unchanged). Notice that this instruction has accumulated a carry.

Special String Sequences: There are two string instructions that have a special meaning. If any of the following instructions: MULAPL, MULSPL, MULTPL, SHLAPL, SHLSPL, SHLTPL, EXTSGNS, MOVAPH immediately precedes ADDS A*n*[~],A*n*[~],PH and SUBS A*n*[~],A*n*[~],PH, the following things happen:

1) Carry generated by the preceding instruction is used in computation.
2) Interrupts *can* occur between these instructions.
3) All instructions in the sequence execute as a single string operation. So, A*n*[**~**] accumulator pointed by the first instruction of the sequence should be used for the remaining instructions in the sequence and changing the value of *n* on one of the above instructions in the sequence has no effect.
4) Accumulators used by ADDS and SUBS (when used with PH) auto–increment internal registers, not AP*n*. So subsequent ADDS and SUBS (immediately following) instructions write into higher accumulators.
5) The sequence ends with ADDS or SUBS (used with PH).
6) These sequences may not give same result when single step debugging because, single stepping changes the internal state. They should be used either with a hardware breakpoint or with fast run mode. The breakpoint should be set after the sequence ends.

For example,
```
MULAPL A0, A0~
ADDS A0, A0, PH
```
The first instruction performs a multiply-accumulate with MR and A0~, and stores PL in A0. The second instruction adds PH to the second word of memory string A0 and puts the result in accumulator string A0~. The MULAPL – ADDS sequence is a special sequence. If A0 is AC0=0xFFFF and MR=0xFF, after execution AC0=0xFF01, AC1=0x00FE. If you replace ADDS A0, A0, PH with ADDS A1, A1, PH and A1 points to a different accumulator, the result is still the same. This is because, the state generated by MULAPL (and other similar instructions described above) is used by ADDS instruction. If another ADDS A0, A0, PH instruction follows the previous one, AC2=0x00FE since the ADDS instruction auto–increments an internal register (not AP*n*). The same reason applies for SUBS A*n*[~],A*n*[~],PH instruction. IMPORTANT: Interrupts may occur between these sequences and the result can be incorrect if the interrupt service changes the state of the processor To prevent interrupts from happening, use the INTD instruction before the execution of the sequence and an INTE afterwards.

4.9 Lookup Instructions

Table lookup instructions transfer data from program memory (ROM) to data memory or accumulators. These instructions are useful for reading permanent ROM data into the user program for manipulation. For example, lookup tables can store initial filter coefficients, characters for an LCD display which can be read for display in the LCD screen, etc. There are four lookup instructions as shown in Table 4–44. Lookup instructions always read the program memory address from the second argument (which is accumulator or its offset). An asterisk (*) always precedes this accumulator to indicate that this is an address.

Table 4–44. Lookup Instructions

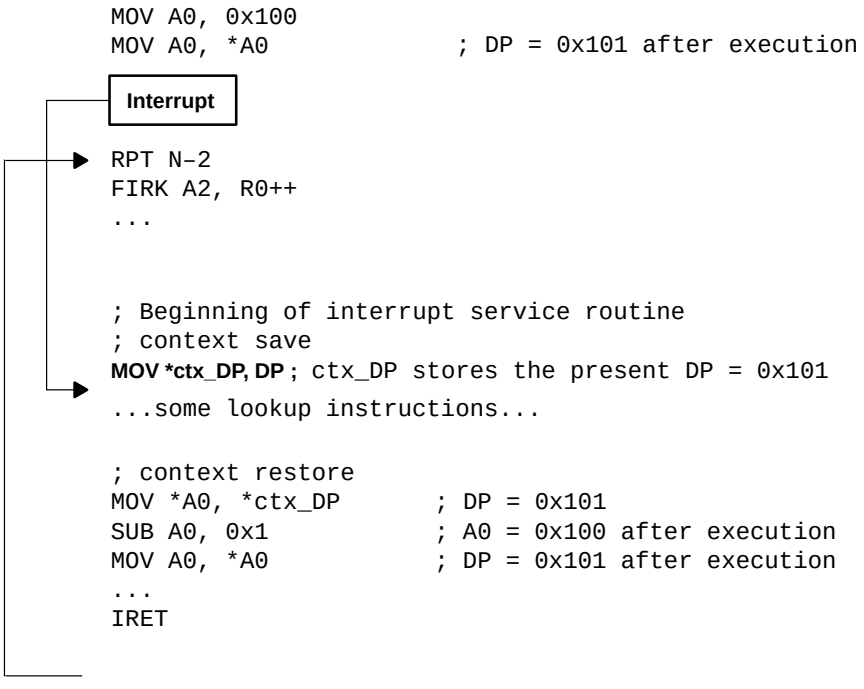
Instructions	Description
	Data Transfer
MOV {*adrs*}, *A*n*	The program memory address is stored in accumulator A*n.* Store the contents of this address in data memory location referred by addressing mode {*adrs*}.
MOV A*n*[~], *A*n*[~] [, *next A*]	The program memory address is stored in accumulator A*n* or its offset A*n*~. Store the contents of this address in accumulator A*n* or A*n*~.
MOVS {*adrs*}, *A*n*	The program memory string address is stored in accumulator A*n.* Store the contents of this address to the data memory string referred by the addressing mode {*adrs*}. The string length is defined in STR register.
MOVS A*n*[~], *A*n*[~]	The program memory string address is stored in accumulator A*n* or its offset A*n*~. Store the contents of this address to the accumulator string A*n* or its offset A*n*~. The string length is defined in STR register.
Data Manipulation on Strings	
ADDS A*n*[~], A*n*[~], *pma16*	ADD the accumulator string A*n* or its offset A*n*~ with the program memory string at location *pma16* and store the result to the accumulator string A*n* or its offset A*n*~. The string length is defined in STR register.
ANDS A*n*[~], A*n*[~], *pma16*	Bitwise/logical AND the string A*n* (or its offset A*n*~) with the program memory string at location *pma16* and store the result in the accumulator string A*n* or its offset A*n*~. The string length is defined in STR register.
CMPS A*n*[~], *pma16*	Compare the accumulator string A*n* (or its offset A*n*~) with the program memory string at location *pma16* and store the result in accumulator string A*n* or its offset A*n*~. The string length is defined in STR register.
SUBS A*n*[~], A*n*[~], *pma16*	Subtract accumulator string A*n* (or its offset A*n*~) with program memory string at location *pma16* and store the result in accumulator string A*n* or its offset A*n*~. The string length is defined in STR register.
XORS A*n*[~], A*n*[~], *pma16*	Bitwise/Logical XOR the accumulator string A*n* or its offset A*n*~ with program memory string at location *pma16* and store the result to accumulator string A*n* or its offset A*n*~. The string length is defined in STR register.

Lookup instructions make use of the data pointer (DP) internally. The DP stores the address of the program memory location, loads the value to the destination, and increments it automatically after every load. Thus, the value of the DP is always the last used program memory address plus one. The content of DP changes after the execution of lookup instructions. If filter instructions FIRK and CORK are used, it is required to context save DP in the interrupt service routine. Since these filter instructions use DP to read coefficient data (see section 4.10), any interrupt occurring between loading the first coefficient and the execution of a FIRK/CORK will change the last value of DP (if the interrupt routine uses a lookup instruction). DP can be stored in RAM (MOV {*adrs*}, DP), and a restoration is done as follows,

```
MOV An, {adrs}
SUB An, 0x1
MOV An, *An
```

Context save and restore of instructions are not required if filter instructions are not used.

Example 4.9.1



4.10 Input/Output Instructions

The MSP50P614/MSP50C614 processor communicates with other on-chip logic as well as external hardware through a parallel I/O interface. Up to 40 I/O ports are addressable with instructions that provide bidirectional data transfer between the I/O ports and the accumulators.

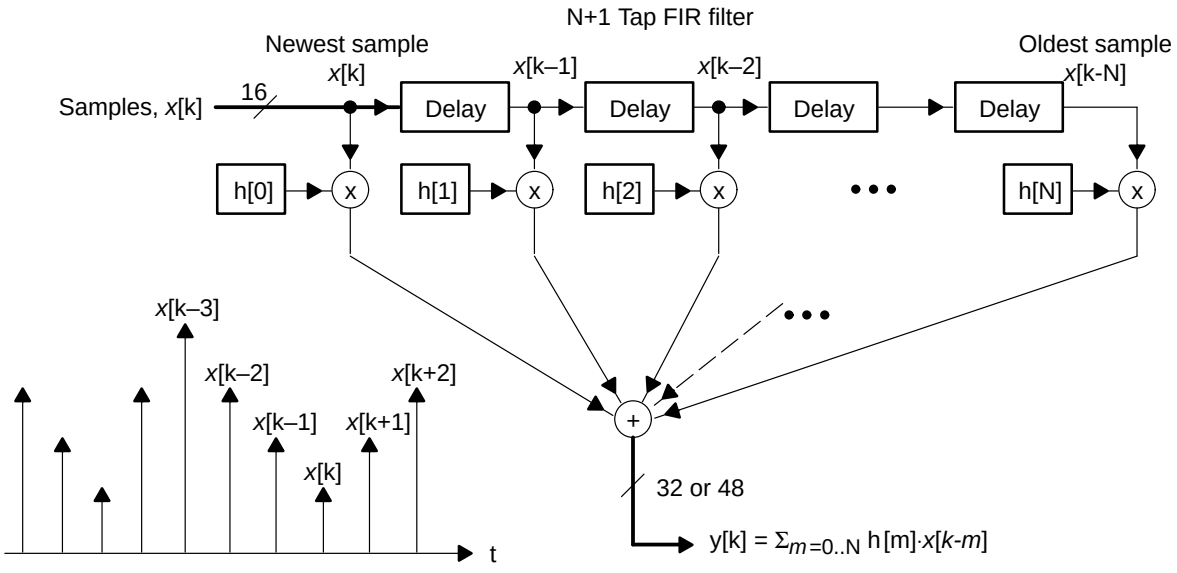
Data input is performed with the IN instruction (Class 6). This instruction uses a memory address and a 4-bit port address. It can also use an accumulator (or offset accumulator) and a 6-bit port address. String transfers are allowed between the accumulators and the input port.

Data output is performed with the OUT instruction (Class 6). The OUT instruction can specify a memory address and a 4-bit port address. It can also use an accumulator (or offset accumulator) and a 6-bit port address. String transfers are allowed between the accumulators and the output port.

4.11 Special Filter Instructions

The MSP50P614/MSP50C614 processor can perform some DSP functions. Fundamental to many filtering algorithms is the FIR structure which requires several parallel operations to execute for each tap of the filter as shown in Figure 4–5. Each tap has 1 multiply and 1 accumulation to obtain the output, y, for N+1 taps,

Figure 4–5. FIR Filter Structure



$$y[k] = h[0]\,x[k] + h[1]x[k\text{-}1] + h[2]x[k\text{-}2] + \cdots + h[N] \times x[k\text{-}N]$$

N tap filters ideally require 2N multiply–accumulates. Four instructions are provided to compute this equation: FIR, FIRK, COR and CORK. All filter instructions require overflow modes to be reset since these instructions have built in overflow hardware. In addition, these instructions must be used with a RPT instruction.

FIR and FIRK instructions perform 16-x-16 bit multiplies and 32-bit accumulation in 2 clock cycles (per tap). The FIR/FIRK instruction takes 2N clock cycles (for N taps) to execute (once inside the RPT loop). FIRK is useful for fixed filters and requires the minimum amount of data memory. However, the DP register may need to be context saved and restored since the filter coefficients are in ROM. FIR is useful for adaptive filtering or applications where coefficients are provided from an external source. FIR does not require a context save and restore for the DP register since both the buffer and the coefficients are in RAM.

COR and CORK instructions perform 16-x-16 bit multiplies and 48-bit accumulation in 3 clock cycles (per tap). Once inside the RPT loop, the total number of clock cycles for an N tap filter is 3N. The COR and CORK instructions are identical in operation and arguments to FIR and FIRK. However, an additional 16-bit extended accumulate cycle is added to prevent the arithmetic overflow common in auto correlation filters.

FIR (COR) Instructions: The execution of the filter instructions is shown in Figure 4–6. To use FIR (COR) instructions, some initial setup is required. Consecutive R*x* pair {Rx_{even}, Rx_{even}+1} should be chosen with Rx_{even} pointing to the RAM sample buffer array and Rx_{even}+1 pointing to the RAM coefficient array. The MR register should be loaded with the first coefficient, *h*[0]. FIR (COR) can now execute with a repeat instruction for N taps. The value of Rx_{even} is incremented during execution. After execution, the last value of Rx_{even} points to the sample buffer location where the next sample can be stored.

FIRK (CORK) Instructions: FIRK (CORK) instructions work exactly the same was as FIR(COR) instructions, however, the coefficient array is located in program memory (ROM). Instead of loading Rx_{even}+1 with the pointer to coefficient array in RAM, the data pointer, DP, is loaded with the value of the coefficient array.

Circular Buffering: The easiest way to understand circular buffering is by example. Suppose a filter, h[n], has three coefficients. Then, theoretically, to calculate one output sample of the filter, the buffer should contain the current sample plus the past 2 samples. Since the output, y[k], for a three tap filter is,

$y[k] = h[0] \bullet \times[k] + h[1] \bullet \times[k-1] + h[2] \bullet \times[k-2]$

On the C614, the circular buffer must contain N+1 samples. In the above example, the buffer must contain four locations (which is one more location than

theory requires). *The second to last RAM location in the circular buffer is tagged* using an STAG instruction. Below is an example of how to set up circular buffering with FIR or COR.

When using the FIR or COR instruction with circular buffering, RAM needs to be allocated for the circular buffer and the filter coefficients. Therefore, the filter coefficient RAM locations must be loaded into RAM and the circular buffer must be cleared before the first FIR or COR instruction is executed.

```
; Set up for FIR filtering (N = 3)
; First clear circular buffer and set tag of second to last
;   sample
   zac    a0
   mov    r0,circBuff   ;point to circular buffer
   rpt    N–2           ;repeat N times
   mov    *r0++,a0      ;clear RAM locations in circular
                        ;  buffer
   mov    *r0,a0        ;N+1 sample in buffer
   mov    r5,2          ;now step back one word and set tag
   sub    r0,r5         ;point r0 back to 2nd to last sample
                        ;  in buffer
   stag   *r0           ;set tag
; Second initialize filter coeffs to proper values
; -----    NOTE: In this code, N must be less than 33 since
; -----          there are only 32 accumulator registers!
   mov    STR,N–2          ;set string length to N
   zacs   a0               ;zero out N accumulators
   mov    a0,FIR_COEFFS;point to filter coeffs
   movs   a0,*a0        ;get N filter coeffs
   mov    r0,coeffs     ;point to RAM locs. for filter coeffs
   movs   *r0,a0        ;put filter coeffs into RAM locs.
   mov    a0,circBuff   ;set up pointer to start of circular
                        ;  buffer
   mov    *startOfBuff,a0
; Initialize filterSTAT_tag (THIS IS IMPORTANT!)
   rovm                 ;This line is MANDATORY!
   sxm                  ;Sample values are signed
   mov    *filterSTAT_tag,STAT
```

Three more details in the above example merit an explanation. The first detail is the pointer to the start of the circular buffer (`startOfBuff`). This keeps track of the location of the newest or current sample in the circular buffer. It moves *backwards* by one location in the buffer each time the FIR or COR instruction is executed so that the oldest sample in the buffer is overwritten with the next sample. This backwards movement is also circular. For example, suppose that `startOfBuff` points to the first RAM location of the circular buffer.

After the FIR or COR instruction executes, the new `startOfBuff` will be the last location in the circular buffer. After another FIR/COR instruction, the new `startOfBuff` will be the second to last location in the circular buffer, and so on.

The second detail is the STAT register. The STAT register must be saved immediately after every FIR or COR instruction. Consequently, this saved value must be loaded before every FIR or COR instruction. If the tag bit in the STAT register is set before an FIR or COR instruction, this tells the processor two things. First, it knows that it must *wrap around* to the first RAM location of the circular buffer. Second, it knows that the `startOfBuff` (and R0) currently points to the last location in the circular buffer. Thus, R0 will increment by R5 after the first multiply. This will become more clear after examining the next example code.

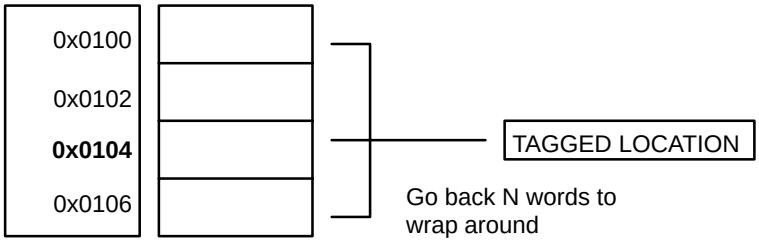
The third detail is that the filter coefficients take up only N RAM locations, but the circular buffer takes up N+1 RAM locations.

Below is an example of the FIR or COR execution inside a DAC interrupt service routine.

```
; FIR Filtering routine (N = 3)
----------------------------------------
   rovm                    ;reset overflow mode
   mov    R5, –2 * N       ;circular buffer length (3 words)
   mov    R1,coeffs        ;R1 points to first of N filter
                              coefficients
   mov    MR,*R1++         ;must increment R1

   mov    R0,*startOfBuff  ;R0 points to start of circular
                              buffer
   mov    AP0,0            ;set up room for the
   mov    STR,0            ; 32 bit output sample (AC0
                              and AC1)
   zacs   A0               ; STR should be 1 for COR/CORK
                              instructions

   mov    STAT,*filterSTAT_tag   ;load STAT with last filter
                                    tag status
   rpt    N–2
   fir    A0,*R0++         ;Do one sample –-> 32 bit result
   mov    *filterSTAT_tag,STAT   ;save STAT with last filter
                                    tag status
                           ;R0 now points to the last/oldest
                              sample
   movs   *ySampleOut,A0   ;FIR outputs bits 0–15 in AC0,
                              16–32 in AC1
```

```
mov    A0,*nextSample   ;Replace last sample with newest
                           sample
mov    *R0,A0           ; and update the start of the
mov    *startOfBuff,R0  ; circular buffer to here (R0)
```

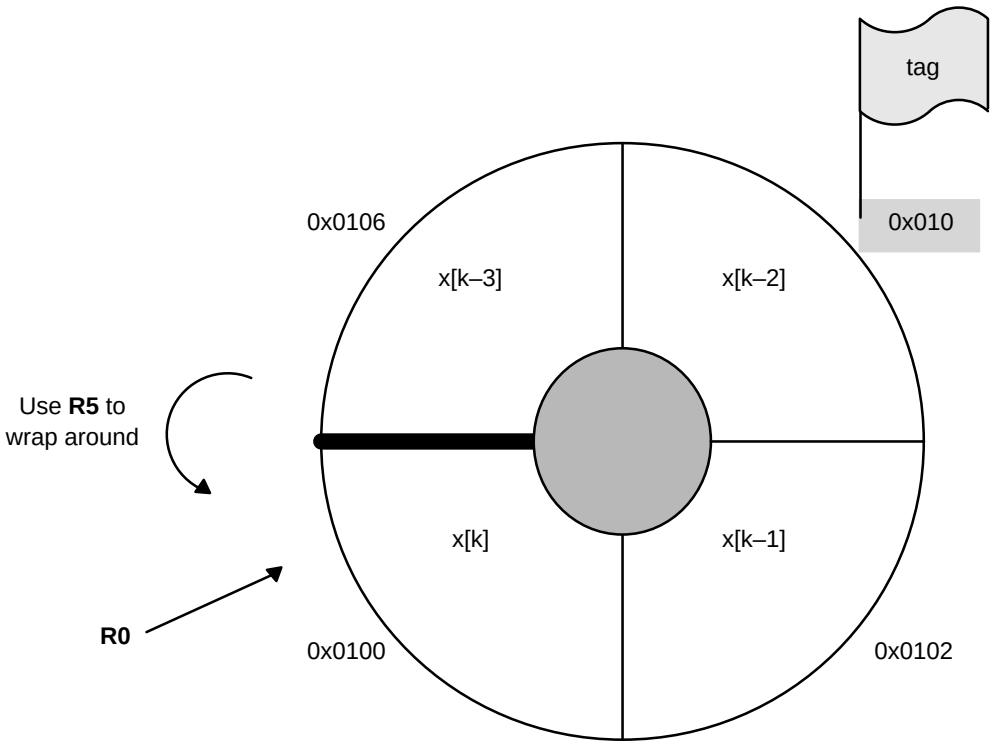
First, the overflow mode must be reset. Next, R5 must be loaded with the *wrap around* value of the circular buffer. *Wrap around* happens automatically. This tells the processor how many words to step back when the end of the circular buffer is reached. This value must be negative and equal to N words even though the buffer is N+1 words long. For example, suppose a four word circular buffer starts at RAM location 0x0100 and ends at 0x0106 (N = 3). In order to wrap around from location 0x0106 back to location 0x0100, the value 0x006 must be subtracted from 0x0106, giving 0x0100.



R0 must point to the current starting point of the circular buffer. R1 must point to the filter coefficients. The MR register must contain the first filter coefficient, h[0]. R0 and R1 must be used this way. The filtering operation will not work if the R*x* registers are reversed. The following are the only allowable register combinations,

R0 points to circular buffer and R1 points to filter coefficients
R2 points to circular buffer and R3 points to filter coefficients

Any combination of registers different from the above will yield incorrect results with the FIR/COR instruction.



After FIR/COR execution

The STAT register is saved in the filterSTAT_tag location. The output of the filtering operation in the example is located in AC0 (lower word) and AC1 (high word). This 32-bit result is stored in the SampleOut RAM location. R0 should be pointing to the oldest sample. The oldest sample, `x[k–3]`, is overwritten by the next sample to be filtered, `x[k+1]`. R0 is saved in the `startOfBuff` pointer for the next FIR/COR instruction

Notice that R0 points backwards by one location from its starting point each time an FIR/COR instruction is executed. In the above figure, R0 would end up at successive locations in a clockwise manner.

Important Note About Setting the STAT Register

It is very important to consider the initial value of the filterSTAT_tag variable. Failure to set up the filterSTAT_tag variable can cause incorrect results in FIR/COR operations. Overflow mode must always be reset. The overflow bit of the STAT register may not be set.

For samples or filter coefficients that are signed, the sign extension mode bit must also be set. Use the following set up for the filterSTAT_tag variable,

```
   rovm  ; Mandatory
; -- Any addition modes can be set hereafter --
   sxm   ; For signed samples, coefficients, filter output
   mov   *filterSTAT_tag,STAT
```

The FIRK/CORK instructions are almost identical to the FIR/COR instructions. The main difference is that the filter coefficients are placed in ROM instead of RAM. In other words, the filter coefficients are in a look-up table. As a result, the R1 register is not used. Before a FIRK/CORK instruction executes, the data pointer register, DP, must be set by the following code,

```
   rovm                  ;reset overflow mode
   mov   R5, -2 * N      ;circular buffer length (3 words)
   mov   A0,FIRK_COEFFS  ; Loads address of lookup table
   mov   A0,*A0          ; Loads first coefficient to A0 and
                            sets DP
   mov   MR,A0           ; Load first coefficient in to MR
                            register
```

In the sequence of code above, the DP register points to the first filter coefficient (in program memory located at FIRK_COEFFS). This happens during the mov A0,*A0 instruction. In addition, the DP register automatically increments to the next address. It should be pointing to the second filter coefficient in program memory. If the contents of the DP register are used somewhere else in the program, a context save and restore must be performed on the DP register for each FIRK/CORK instruction. See the chapter 4 section called, Lookup Instructions. During FIRK/CORK execution, the MR register is loaded with the contents of the DP register, the DP register increments, pointing to the next filter coefficient, and the multiply-accumulate is performed.

The remaining FIRK/CORK code is almost the same as the FIR/COR code.

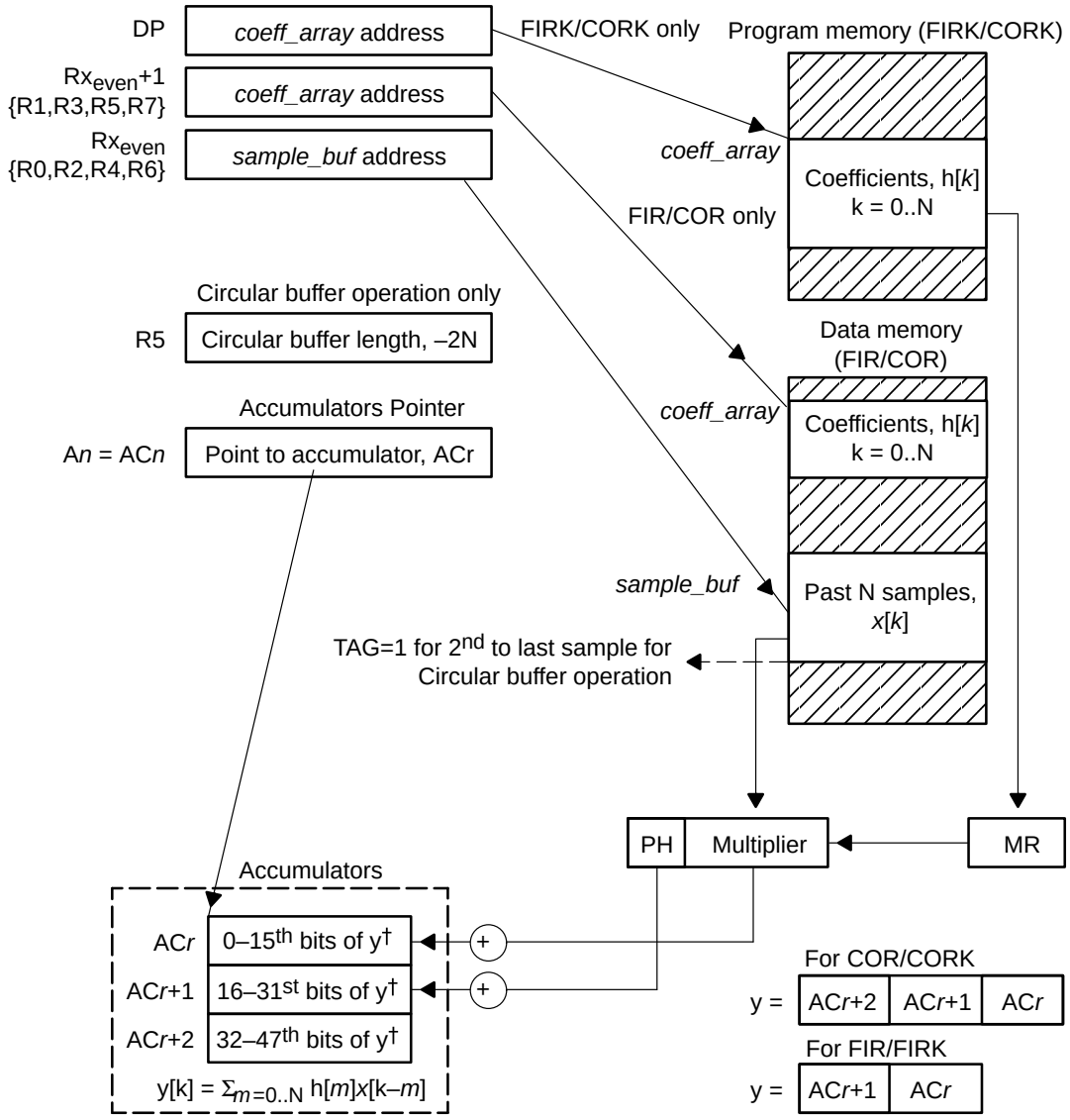
```
   mov   R0,*startOfBuff ;R0 points to start of circular
                            buffer
   mov   AP0,0           ;set up room for the
   mov   STR,0           ;32 bit output sample (AC0 and
                            AC1)
   zacs  A0              ; STR should be 1 for COR/CORK
                            instructions
```

```
mov    STAT,*filterSTAT_tag    ;load STAT with last filter
                                tag status
rpt    N–2
firk   A0,*R0++             ;Do one sample –-> 32 bit result
mov    *filterSTAT_tag,STAT    ;save STAT with last filter
                                tag status
                               ;R0 now points to the last
                                sample
movs   *ySampleOut,A0          ;FIR outputs bits 0–15 in
                                AC0, 16–32 in AC1

mov    A0,*nextSample          ;Replace last sample with
                                 newest sample and update
mov    *R0,A0                  ; the start of the
mov    *startOfBuff,R0         ; circular buffer to here
                                 (R0)
```

The set up for the FIRK/CORK instruction is the same as the set up for the FIR/COR instruction with the exception that the filter coefficients do not need to be loaded into RAM locations. Rather, they can be included just before speech data or elsewhere in the program code as follows,

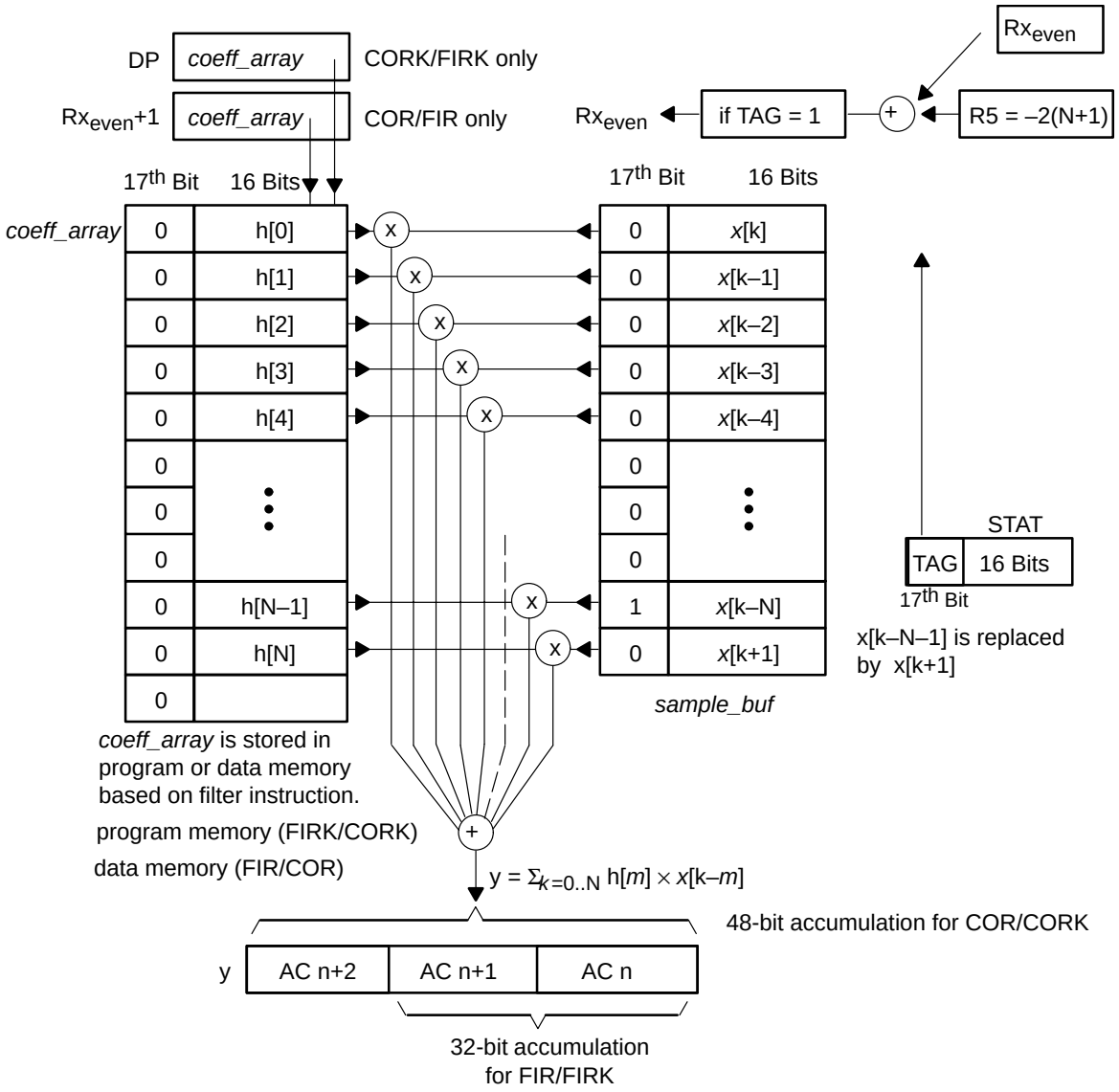
```
FIRK_COEFFS
   include   “\..\tables\coeffs.dat”
```

Figure 4–6. Setup and Execution of MSP50P614/MSP50C614 Filter Instructions, N+1 Taps



† The value of y is stored in AC*r* and AC*r*+1 for FIR instruction (32-bit accumulation). COR instruction uses 48-bit accumulation and includes accumulator AC*r*+2.

Figure 4–7. Filter Instruction and Circular Buffering for N+1 Tap Filter



4.12 Conditionals

The condition bits in the status register (STAT) are used to modify program control through conditional branches and calls. Various combinations of bits are available to provide a rich set of conditional operations. These condition bits can also be used in Boolean operations to set the test flags TF1 and TF2 in the status register.

STAT register bit settings	**Arithmetic/Logic Condition**	**Condition mnemonic**	**Alternate† mnemonic**	**'NOT'‡ condition mnemonic**	**'NOT' ‡ condition alternate† mnemonic**
ZF = 1	Zero flag	ZF		NZF	
SF = 1	Sign flag	SF		NSF	
CF = 1	Carry flag	CF		NCF	
ZF = 0 & CF = 0	Below (unsigned)	B	NAE	NB	AE
ZF = 0 & CF = 1	Above (unsigned)	A	NBE	NA	BE
ZF = 1 & SF = 0	Greater (signed)	G	NLE	NG	LE
ZF = 1 & OF = 0	Equal	E		NE	
OF = 1	Overflow flag	OF		NOF	
ZF = 0 & SF = 1	Less (signed)	L	NGE	NL	GE
RCF = 1	R*x* carry flag	RCF		RNCF	
RZF = 0 & RCF = 1	R*x* above (unsigned)	RA	RNBE	RNA	RBE
RZF = 1	R*x* equal	RE	RZ	RNE	RNZ
TF1 = 1	Test flag 1	TF1		NTF1	
TF2 = 1	Test flag 2	TF2		NTF2	
TAG = 1	Memory tag	TAG		NTAG	
IN1§	Input line 1	IN1		NIN1	
IN2§	Input line 2	IN2		NIN2	
XZF = 1	Transfer zero flag	XZF		XNZF	
XSF = 1	Transfer sign flag	XSF		XNSF	
XZF = 0 & XSF = 0	Transfer greater (signed)	XG	XNLE	XNG	XLE

† Alternate mnemonics are provided to help program readability. They generate the same opcodes as the associated condition.
‡ Status register (STAT) bit settings are inverted for NOT conditions.
§ Hardware lines used for I/O expansion design. These lines are PA0 and PA1.

4.13 Legend

All instructions of the MSP50P614/MSP50C614 use the following syntax:

name [*dest*] [, *src*] [, *src1*] [, *mod*]

name	Name of the instruction. Instruction names are shown in bold letter through out the text.
dest	Destination of the data to be stored after the execution of the instruction. Optional for some instructions or not used. Destination is also used as both source and destination for some instructions.
src	Source of the first data. Optional for some instructions or not used.
src1	Source of the second data. Some instructions use a second data source. Optional for some instructions or not used.
mod	Post modification of a register. This can be either *next A* or *Rmod* and will be specified in the instruction.

The following table describes the meanings of the symbols used in the instruction set descriptions:

Bold type means it must be typed exactly as shown.
italics type means it is a variable.
[ ] square brackets enclose optional arguments.

Operands

0 ≤ *dma6* ≤ 63	
0 ≤ *dma16* ≤ 65535	dma16 ≤ 639 for MSP50P614/MSP50C614
0 ≤ *imm5* ≤ 31	
0 ≤ *imm16* ≤ 65535	
0 ≤ *offset6* ≤ 63	
0 ≤ *offset7* ≤ 127	
0 ≤ *offset16* ≤ 65535	
0 ≤ *pma8* ≤ 255	
0 ≤ *pma16* ≤ 65535	pma16 ≤ 32767 for MSP50P614/MSP50C614
0 ≤ *port4* ≤ 15	
0 ≤ *port6* ≤ 63	

Symbol	Meaning
!	Invert the bit of the source. Used with flag addressing only.
A*n*	Accumulator selector where *n* = 0...3. A*n* is the accumulator pointed by AP*n*.
A*n*~	Offset accumulator selector where n = 0...3. A*n* is the accumulator pointed by AP*n*+16; AP*n* wraps after 31.

Symbol	Meaning
A~	Select offset accumulator as the source if this bit is 1. Used in opcode encoding only.
~A	Select offset accumulator as the destination accumulator if this bit is 1. Used in opcode encoding only.
A~	Select offset accumulator as the source if this bit is 0. Used in opcode encoding only.
~A~	Can be either ~A or A~ based on opcode (or instruction). Used in Opcode encoding only.
A*n*[~]	Can be either A*n* or A*n*~ where *n* = 0...3
AP*n*	Accumulator Pointer register where *n* = 0..3. Low-order 5 bits select one of 32 accumulators.
adrs	Addressing mode bits *am*, R*x*, *pm.* See Table 4–46.
{*adrs*}$_n$	*Addr*essing mode which must be provided. It should be of the format shown in Table 4–46. The curly braces {} are not included in the actual instruction. The subscript *n* represents the data size (in bits) the instruction will use. For example, {adrs}$_8$ means that the instruction will use 8-bit data from the addressed memory and the upper bits may not be used. If *n* is not provided, data width is 16 bits.
cc	Condition code bits used with conditional branch/calls and test flag/bit instructions.
{*cc*}	Conditional code mnemonic used with conditional branch/calls and test flag/bit instructions. Curly braces indicates this field is not optional.
CF	Carry flag
clk	Total clock cycles per instruction
dma[*n*]	*n* bit data memory address. For example, *dma8* means 8-bit location data memory address. If *n* is not specified, defaults to *dma16*.
DP	Data pointer register, 16 bits
flagadrs	Flag addressing syntax as shown in Table 4–47.
flg	Test flag bit. Used in opcode encoding only.
{*flagadrs*}	Flag addressing syntax as shown in Table 4–48.
FM	Fractional mode
g/r	Global/relative flag bit for flag addressing.
IM	Interrupt enable mode
imm[*n*]	*n* bit immediate value. If *n* is not specified, defaults to *imm16*.
k0...kn	Constant field bits.
MR	Multiply register, 16 bits
next A	Accumulator pointer premodification. See Table 4–45.
Not	Not condition on conditional jumps, conditional calls or test flag instructions.
N/R	Not repeatable or not recommended

Symbol	Meaning
n_R	Value in repeat counter loaded by RPT instructions
n_S	Value in string register STR
OF	Overflow flag
offset[*n*]	*n* bit offset from a reference register.
OM	Overflow mode
PC	Program counter, 16 bits
pma[*n*]	*n* bit program memory address. For example, pma8 means 8-bit program memory address. If *n* is not specified, defaults to *pma16*.
port[*n*]	*n* bit I/O port address. Certain instructions multiply this port address by 4.
PH	Product high register, 16 bits
PL	Product low register, 16 bits (cannot be read/written directly)
R	R*x* register treated as a general purpose register. This bit is not related to any addressing mode.
RCF	Register carry flag
R*x*	Indirect register *x* where *x* = 0..7
RZF	Register zero flag
s	Represents string mode if 1, otherwise normal mode.
SF	Sign flag
STAT	Status register, 17 bits
STR	String register, 8 bits
SV	Shift value register, 4 bits
TAG	Memory tag
TF1	Test flag 1
TF2	Test flag 2
TOS	Top of stack register, 16 bits
UM	Unsigned mode
w	Word(s) taken by instruction
x	Don’t care
XM	Extended sign mode
XSF	Transfer (TX) sign flag
XZF	Transfer (TX) zero flag
ZF	Zero flag

Table 4–45. Auto Increment and Decrement

Operation	*next A*	**b9**	**b8**
No modification		0	0
Auto increment	++A	0	1
Auto Decrement	−−A	1	0

Table 4–46. Addressing Mode Bits and adrs Field Description

Relative Addressing Modes	**Clocks** *clk*	**Words** *w*	**String† Repeat Operation Clocks**	{*adrs*}	**Addressing Mode Encoding**							
					7	6	5	4	3	2	1	0
					am			Rx (x = 0 … 7)			*pm*	
Direct	2	2	n_R+4	**dma16*	0	0	0	x			0	0
Short relative	1	1	n_R+2	*R6 + *offset7*	1	*offset7*						
Relative to R5	1	1	n_R+2	*R*x* + R5	0	1	0	R*x*			0	0
Long relative	2	2	n_R+4	*R*x* + *offset16*	0	0	1	R*x*			0	0
Indirect	1	1	n_R+2	*R*x*	0	1	1	R*x*			0	0
				*R*x*++							0	1
				*R*x*−−							1	0
				*R*x*++R5							1	1

† Replace n_R with n_S for string operation.

Note: *dma16* and *offset16* is the second word.

Table 4–47. Flag Addressing Syntax and BIts

Flag Addressing Modes	**Clocks** *clk*	**Words** *w*	**Repeat Operation†** *clk*	{*flagadrs*} **Syntax**	**flag addressing mode encoding,** *flagadrs*						
					6	5	4	3	2	1	0
					flag address bits						*g/r*
Global	1	1	n_R+2	**dma6*	*dma6*						0
Relative	1	1	n_R+2	*R6+*offset6*	*offset6*						1

† n_R is RPT instruction argument

4.14 Individual Instruction Descriptions

In this section, individual instructions are discussed in detail. Use the conditionals in Section 4.12 and the legend in Section 4.13 to help with individual instruction descriptions. Each instruction is discussed in detail and provides the following information:

- Assembler syntax
- Clock cycles required with or without repeat instructions
- Words required
- Limitation and restrictions
- Execution
- Affected flags
- Opcode
- Description
- Recommendation to other related instructions (See Also field)
- Examples

4.14.1 ADD Add word

Syntax

[*label*]	**name**	*dest*, *src* [, *src1*] [,*mod*]	**Clock**, *clk*	**Words**, *w*	**With RPT,** *clk*	**Class**
	ADD	A*n*[~]**,** A*n***,** {*adrs*} [**,** *next A*]	Table 4–46	Table 4–46	Table 4–46	1a
	ADD	A*n*[~]**,** A*n*[~]**,** *imm16* [**,** *next A*]	2	2	N/R	2b
	ADD	A*n*[**~**]**,** A*n*[**~**]**,** PH [**,** *next A*]	1	1	n_R+3	3
	ADD	A*n*[**~**]**,** A*n*~**,** A*n* [**,** *next A*]	1	1	n_R+3	3
	ADD	R*x***,** *imm16*	2	2	N/R	4c
	ADD	R*x***,** R5	1	1	n_R+3	4d
	ADD†	AP*n***,** *imm5*	1	1	N/R	9c

† Does not affect the status flags.

Execution

[premodify AP if *mod* specified]
dest ⇐ *dest* + *src* (for two operands)
dest ⇐ *src* + *src1* (for three operands)
PC ⇐ PC + *w*

Flags Affected

dest is A*n*:	OF**,** SF**,** ZF**,** CF are set accordingly
dest is R*x*:	RCF**,** RZF are set accordingly
src1 is {*adrs*}:	TAG is set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD A*n*[~], A*n*, {*adrs*} [, *next A*]	0	0	0	0	~A	*next A*		A*n*		*adrs*							
	x	*dma16 (for direct) or offset16 (long relative)* [see section 4.13]															
ADD A*n*[~], A*n*[~], *imm16* [, *next A*]	1	1	1	0	0	*next A*		A*n*		0	0	0	0	0	1	A~	~A
	x	*imm16*															
ADD A*n*[~], A*n*[~], PH [, *next A*]	1	1	1	0	0	*next A*		A*n*		0	1	1	0	1	0	A~	~A
ADD A*n*[~], A*n*~, A*n* [, *next A*]	1	1	1	0	0	*next A*		A*n*		0	0	1	0	1	0	A~	~A
ADD R*x*, *imm16*	1	1	1	1	1	1	1	0	0	0	0	0	R*x*			0	0
	x	*imm16*															
ADD R*x*, R5	1	1	1	1	1	1	1	0	0	1	0	0	R*x*			0	0
ADD AP*n*, *imm5*	1	1	1	1	1	0	1	AP*n*		0	1	0	*imm5*				

Description

Syntax	Description
ADD *dest*, *src*	ADD *src* with *dest* and store the result to *dest*.
ADD *dest*, *src*, *src1* [,*mod*]	ADD *src1* with *src* and store the result to *dest*. Premodify the *mod* before execution. (if provided)

See Also ADDB, ADDS, SUB, SUBB, SUBS

Example 4.14.1.1 `ADD A2~, A2, *R2++R5, --A`
Decrement accumulator pointer AP2. Add word at address in R2 to A2, put result in A2**~**. Add value in R5 to R2 and store in R2.

Example 4.14.1.2 `ADD A1, A1, 0x1221`
Add immediate value of 0x1221 to A1 and store result in A1.

Example 4.14.1.3 `ADD A0, A0~, PH`
Add PH to accumulator A0**~** and store result in accumulator A0.

Example 4.14.1.4 `ADD A1, A1~, A1`
Add accumulator **A1** to accumulator A1**~**, put result in accumulator A1.

Example 4.14.1.5 `ADD R3, 0x1000`
Add 0x1000 to register R3 store result in R3.

Example 4.14.1.6 `ADD R2, R5`
Add R2 to R5, store result in R2.

Example 4.14.1.7 `ADD AP3, 0x10`
Add immediate 0x10 to accumulator pointer AP3, store result in accumulator pointer AP3.

4.14.2 ADDB ADD BYTE

Syntax

[*label*]	***name***	*dest, src*	**Clock**, *clk*	**Words**, *w*	**With RPT**, *clk*	**Class**
	ADDB	A*n*, *imm8*	1	1	N/R	2a
	ADDB	R*x*, *imm8*	1	1	N/R	4b

Execution
dest ⇐ *dest* + *src*
PC ⇐ **PC** + 1

Flags Affected
dest is **A***n*: OF**,** SF**,** ZF**,** CF are set accordingly
dest is **R***x*: RCF**,** RZF are set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADDB A*n imm5*	1	0	1	0	0	0	0	A*n*		*imm8*							
ADD R*x*, *imm8*	1	0	1	1	0	0	*k4*	*k3*	*k2*	*k7*	*k6*	*k5*	R*x*			*k1*	*k0*

See Also ADD, ADDS, SUB, SUBB, SUBS

Description Add immediate value of unsigned *src* byte to value stored in *dest* register and store result in the same *dest* register.

Example 4.14.2.1 `ADDB A2, 0x45`
Add immediate 0x45 to A2.

Example 4.14.2.2 `ADDB R5, 0xf2`
Add immediate 0xf2 to R5.

4.14.3 ADDS Add String

Syntax

[*label*]	***name***	*dest, src, src1*	**Clock**, *clk*	**Words**, *w*	**With RPT**, *clk*	**Class**
	ADDS	A*n*[~], A*n*, {*adrs*}	Table 4–46	Table 4–46	Table 4–46	1a
	ADDS	A*n*[~], A*n*[~], *pma16*	n_S+4	2	N/R	2b
	ADDS	A*n*[~], A*n*~, A*n*	n_S+2	1	n_R+2	3
	ADDS†	A*n*[~], A*n*[~], PH	1	1	1	3

† This instruction ignores the string count, executing only once but maintains the CF and ZF status of the previous multiply or shift operation as if the sequence was a single string. This instruction should immediately follow one of the following class 1b instructions: MOVAPH, MULAPL, MULSPL, SHLTPL, SHLSPL, and SHLAPL. An interrupt *should not* occur between one of these instructions and ADDS. An interrupt may cause incorrect results. Interrupts must be explicitly disabled at least one instruction before the class 1b instruction. This special sequence is protected inside a BEGLOOP – ENDLOOP construct. In addition, single stepping is not allowed for this instruction. A*n* in this instruction should be the same as A*n* in one of the listed class 1b instruction. Offsets are allowed. See Section 4.8 for more detail.

Execution *dest* string ⇐ *src* string + *src1* string
PC ⇐ PC + *w*

Flags Affected *dest* is **A***n*: **OF, SF, ZF, CF** are set accordingly
src1 is {*adrs*}: TAG is set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADDS A*n*[~], A*n*, {*adrs*}	0	0	0	0	~A	1	1	A*n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
ADDS A*n*[~], A*n*[~], *pma16*	1	1	1	0	0	1	1	A*n*		0	0	0	0	0	1	A~	~A
	x	pma16															
ADDS A*n*[~], A*n*~, A*n*	1	1	1	0	0	1	1	A*n*		0	0	1	0	1	0	A~	~A
ADDS A*n*[~], A*n*[~], PH	1	1	1	0	0	1	1	A*n*		0	1	1	0	1	0	A~	~A

Description Add value of *src* string to the value of *src1* string and store resulting string in *dest*. String length minus two should be stored in STR before execution.

See Also ADD, ADDB, SUB, SUBB, SUBS

Example 4.14.3.1 `ADDS A0, A0~, *R2`
Add data memory string beginning at address in R2 to accumulator string A0~, put result in accumulator string A0.

Example 4.14.3.2 `ADDS A0, A0~, 0x1400`
Add program memory string beginning at address 0x1400 to accumulator string A0~, put result in accumulator string A0.

Example 4.14.3.3 `ADDS A1, A1~, A1`

Add accumulator string A1 to accumulator string A1**~**, put result in accumulator string A1.

Example 4.14.3.4
```
MULAPL A0, A0~
ADDS A0, A0~, PH
```

The first instruction multiplies MR and A0~, adds PL to A0, and stores the result in A0. The second instruction adds PH to the second word of memory string A0 and puts the result in accumulator string A0. Note that MULAPL and ADDS constitute a special sequence. When this sequence occurs, interrupts are NOT disabled, so interrupts should be disabled for correct operation. In extended sign mode, if A0 is AC0 = 0x0000, A0~ is AC16=0xFFFF and MR=0xFF, after execution AC0=0xFF01, AC1=0xFFFF.

4.14.4 AND Bitwise AND

Syntax

[label]	***name***	*dest, src* [*, src1*] [*, mod*]	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	AND	A*n*, {*adrs*}	Table 4–46		Table 4–46	1b
	AND	A*n*[~]**,** A*n*[~], *imm16* [, *next A*]	2	2	N/R	2b
	AND	A*n*[~], A*n*~, A*n* [, *next A*]	1	1	n_R+3	3
	AND	TF*n*, [!]{*flagadrs*}	1	1	N/R	8a
	AND	TF*n*, {*cc*} [, R*x*]	1	1	n_R+3	8b

Execution

[premodify AP if *mod* specified]
dest ⇐ *dest* AND *src* (for two operands)
dest ⇐ *src* AND *src1* (for three operands)
PC ⇐ PC + *w*

Flags Affected

dest is A*n*: OF**,** SF**,** ZF**,** CF are set accordingly
dest is TF*n*: TF*n* bits in STAT register are set accordingly
src is {*adrs*}: TAG bit is set accordingly
src is {*flagadrs*}: TAG bit is set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AND A*n*, {*adrs*}	0	1	0	0	0	1	0	A*n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
AND A*n*[~], A*n*[~], *imm16* [*, next A*]	1	1	1	0	0	*next A*		A*n*		1	0	1	0	0	1	A~	~A
	x	*imm16*															
AND A*n*[~], A*n*~, A*n* [*, next A*]	1	1	1	0	0	*next A*		A*n*		0	1	0	1	0	0	A~	~A
AND TF*n*, {*flagadrs*}	1	0	0	1	1	*flg*	*Not*	1	0	0	*flagadrs*						
AND TF*n*, {*cc*} [, R*x*]	1	0	0	1	0	*flg*	*Not*	*cc*				R*x*				1	0

Description

Syntax	**Description**
AND *dest*, *src*, *src1* [, *mod*]	Bitwise AND *src1* and *src* and store result in *dest*. Premodification of accumulator pointers are allowed with some operand types.
AND *dest*, *src*	Bitwise AND *dest* and *src* and store result in *dest*.
AND TF*n*, {*flagadrs*}	AND TF*n* bit with 17th bit of data memory address referred by addressing mode {*flagadrs*}, store result in TF*n* bit in STAT register. *n* is either 1 or 2.
AND TF*n*, {*cc*} [, R*x*]	AND test condition {*cc*} with TF*n* bit in STAT register. R*x* must be provided if *cc* is one of {RZP**,** RNZP**,** RLZP**,** RNLZP} to check if the selected R*x* is zero or negative. R*x* should not be provided for other conditionals. *n* is 1 or 2.

See Also ANDS, ANDB, OR, ORB, ORS, XOR, XORB, XORS

Example 4.14.4.1 `AND A3, *R4––`
And word at address in R4 to A3, store result in A3. Decrement value in R4 by 2 (word mode) after the AND operation.

Example 4.14.4.2 `AND A0~, A0, 0xff0f, ––A`
Predecrement accumulator pointer AP0. And immediate value 0xff0f to register accumulator A0, store result in accumulator A0~.

Example 4.14.4.3 `AND TF2, *0x0020`
AND global flag bit at RAM word location 0x0020 to TF2 in the STAT. Store result in the TF2 bit in the STAT register. Note that {flagadrs} cannot exceed values greater than *0x003F.

Example 4.14.4.4 `AND TF1, TF2`
AND TF1 with TF2 bit in the STAT register and store result in TF1.

4.14.5 ANDB Bitwise AND Byte

Syntax

[label]	***name***	*dest, src*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	ANDB	A*n*, *imm8*	1	1	N/R	2a

Execution *dest* ⇐ *dest* AND *src* byte
PC ⇐ PC + 1

Flags Affected **OF**, **SF**, **ZF**, **CF** are set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ANDB A*n*, *imm8*	1	0	1	0	1	0	1	A*n*		*imm8*							

Description Bitwise AND *src* byte and byte stored in *dest* register and store result in *dest* register.

See Also AND, ANDS, OR, ORB, ORS, XOR, XORB, XORS

Example 4.14.5.1
```
ANDB A2, 0x45
```
AND immediate value 0x45 to A2 (byte mode). Store result in A2. Upper 8 bits of A2 will be ANDed with zeros.

4.14.6 ANDS Bitwise AND String

Syntax

[label]	***name***	*dest, src* [, *src1*]	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	ANDS	A*n*, {*adrs*}	Table 4–46		Table 4–46	1b
	ANDS	A*n*[**~**], A*n*[**~**]**,** *pma16*	n_R+4	1	N/R	2b
	ANDS	A*n*[**~**], A*n***~**, A*n*	n_R+3	1	n_R+3	3

Execution

dest string ⇐ *dest* string AND *src* string (for two operands)
dest string ⇐ *src* string AND *src1* string (for three operands)
PC ⇐ PC + *w*

Flags Affected

dest is **A***n*: OF**,** SF**,** ZF**,** CF are set accordingly
src is {*adrs*}: TAG bit is set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ANDS A*n*, {*adrs*}	0	1	0	0	0	1	1	A*n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
ANDS A*n*[~], A*n*[~], *pma16*	1	1	1	0	0	1	1	A*n*		1	0	1	0	0	1	A~	~A
	x	*pma16*															
ANDS A*n*[~], A*n*~, A*n*	1	1	1	0	0	1	1	A*n*		0	1	0	1	0	0	A~	~A

Description

Syntax	**Description**
ANDS *dest*, *src*	Bitwise AND of *src* string and *dest* string and store result in *dest* string.
ANDS *dest*, *src, src1*	Bitwise AND *src1* string *src* string and store result in *dest* string.

See Also AND, ANDB, OR, ORB, ORS, XOR, XORB, XORS

Example 4.14.6.1
```
ANDS A0, *R2
```
AND data memory string beginning at address in R2 to A0, put result in A0.

Example 4.14.6.2
```
ANDS A0~, A0, 0x1400
```
AND program memory string beginning at address in 0x1400 to A0, put result in A0**~**.

Example 4.14.6.3
```
ANDS A0, A0~, A0
```
AND accumulator string A0 to accumulator string A0**~**, put result in accumulator string A0.

Example 4.14.6.4
```
ANDS A0, A0~, *R2
```
AND memory string beginning at address in R2 to A0**~**, put result in A0.

4.14.7 BEGLOOP Begin Loop

Syntax

[label]	***name***	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	BEGLOOP†	1	1	N/R	9d

† Loop must end with **ENDLOOP**.

Execution

Save next instruction address (PC + 1)
(*mask interrupts*)
PC ⇐ PC + 1

Flags Affected **none**

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BEGLOOP	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0

Description

This instruction saves the next sequential address in a shadow register and masks interrupts. Interrupts occurring during execution of this and following instructions are actually queued until the loop is complete (see ENDLOOP). The loop executes N number of times. Thus, N – 2, should be loaded in R4 in order to loop N times.

BEGLOOP and ENDLOOP block has following restrictions:

- ❏ No CALL instructions can be used.
- ❏ All maskable interrupts are queued.
- ❏ BEGLOOP/ENDLOOP block cannot be nested.

See Also ENDLOOP

Example 4.14.7.1

```
MOV R4, count – 2 ;init R4 with loop count
BEGLOOP
ADD A0, A0~, A0   ;add A0~ to A0 (count) times
ENDLOOP
```

Initialize R4 with the loop count value minus 2 to repeat the loop for count times. Execute the ADD A0, A0~, A0 instruction until R4 is negative. R4 is decremented each time ENDLOOP is encountered. When R4 is negative, ENDLOOP becomes a NOP and execution continues with the next instruction after ENDLOOP.

4.14.8 CALL Unconditional Subroutine Call

Syntax

[label]	***name***	*address*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	CALL	*pma16*	2	2	N/R	7c
	CALL	***A**n*	2	1	N/R	7c

Execution

R7 ⇐ R7 + 2
*R7 ⇐ TOS
TOS ⇐ PC + 2
PC ⇐ ***A**n or pma16

Flags Affected None

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CALL *pma16*	1	0	0	0	0	1	0	1	0	1	0	1	1	1	1	1	0
	x	*pma16*															
CALL *A*n*	1	0	0	0	1	1	0	A*n*†		0	0	0	1	1	1	1	0

† The value of An is in the following table.

An	Bit 9	Bit 8
A0	0	0
A1	0	1
A2	1	0
A3	1	1

Description PC + w is pushed onto the top of stack (TOS) and the second word operand or accumulator value is loaded into the PC. Call instructions cannot immediately followed by RET instructions. No restrictions apply if IRET is used instead of RET.

Syntax	Description
CALL *pma16*	Unconditional call to specified program memory address *pma16*.
CALL *A*n*	Call to address referenced by A*n*.

See Also C*cc*, VCALL, RET, IRET

Example 4.14.8.1 `CALL 0x2010`
Call unconditionally program memory address 0x2010.

Example 4.14.8.2 `CALL *A0`
Call unconditionally program memory address stored in accumulator A0.

Note:

You can not RET to a RET. For example, the following code can cause problems:

```
CALL   my sub
RET
```

To eliminate any problem, a NOP (or other code) should be inserted between the CALL and the RET. For example:

```
CALL   my sub
NOP
RET
```

4.14.9 C*cc* Conditional Subroutine Call

Syntax

[label]	***name***	***address***	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	Ccc†	*pma16*	2	2	N/R	7c

† Cannot immediately follow a **CALL** instruction with a return instruction.

If true			If Not true		
[*label*]	CZ	*pma16*	[*label*]	CNZ	*pma16*
[*label*]	CS	*pma16*	[*label*]	CNS	*pma16*
[*label*]	CC	*pma16*	[*label*]	CNC	*pma16*
[*label*]	CG	*pma16*	[*label*]	CNG	*pma16*
[*label*]	CE	*pma16*	[*label*]	CNE	*pma16*
[*label*]	CA	*pma16*	[*label*]	CNA	*pma16*
[*label*]	CB	*pma16*	[*label*]	CNB	*pma16*
[*label*]	CO	*pma16*	[*label*]	CNO	*pma16*
[*label*]	CRC	*pma16*	[*label*]	CRNC	*pma16*
[*label*]	CRE	*pma16*	[*label*]	CRNE	*pma16*
[*label*]	CL	*pma16*	[*label*]	CNL	*pma16*
[*label*]	CTF1	*pma16*	[*label*]	CNTF1	*pma16*
[*label*]	CTF2	*pma16*	[*label*]	CNTF2	*pma16*
[*label*]	CTAG	*pma16*	[*label*]	CNTAG	*pma16*
[*label*]	CIN1	*pma16*	[*label*]	CNIN1	*pma16*
[*label*]	CIN2	*pma16*	[*label*]	CNIN2	*pma16*
[*label*]	CXZ	*pma16*	[*label*]	CXNZ	*pma16*
[*label*]	CXS	*pma16*	[*label*]	CXNS	*pma16*
[*label*]	CXG	*pma16*	[*label*]	CXNG	*pma16*
[*label*]	CRA	*pma16*	[*label*]	CRNA	*pma16*

Execution

IF (*cc* = true)
***R7** ⇐ **TOS**

TOS ⇐ **PC** + 2
PC ⇐ *pma16*
R7 ⇐ **R7** + 2
ELSE
NOP
PC ⇐ **PC** + 2

Flags Affected **none**

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C*cc pma16*	1	0	0	0	0	1	*Not*	*cc*					0	0	0	0	0
	x	*pma16*															

Table 4–48. Names for cc

cc					*cc* names		**Description** **True condition (*Not* true condition)**
					cc name	*Not cc* name	
0	0	0	0	0	Z	NZ	Conditional on ZF=1 (*Not* condition ZF=0)
0	0	0	0	1	S	NS	Conditional on SF=1 (*Not* condition SF=0)
0	0	0	1	0	C	NC	Conditional on CF=1 (*Not* condition CF=0)
0	0	0	1	1	B	NB	Conditional on ZF=0 and CF=0 (*Not* condition ZF≠0 or CF≠0)
0	0	1	0	0	A	NA	Conditional on ZF=0 and CF=1 (*Not* condition ZF≠0 or CF≠1)
0	0	1	0	1	G	NG	Conditional on SF=0 and ZF=0 (*Not* condition SF≠0 or ZF≠0)
0	0	1	1	0	E	NE	Conditional if ZF=1 and OF=0 (*Not* condition ZF≠1 or OF≠0)
0	0	1	1	1	O	NO	Conditional if OF=1 (*Not* condition OF=0)
0	1	0	0	0	RC	RNC	Conditional on RCF=1 (*Not* condition RCF=0)
0	1	0	0	1	RA	RNA	Conditional on RZF=0 and RCF=1 (*Not* condition RZF≠0 or RCF≠1)
0	1	0	1	0	RE	RNE	Conditional on RZF=1 (*Not* condition RZF=0)
0	1	0	1	1	RZP	RNZP	Conditional on value of R*x*=0 Not available on calls. (*Not* condition R*x*≠0)
0	1	1	0	0	RLZP	RNLZP	Conditional on MSB of Rx=1. Not available on calls. (*Not* condition MSB of R*x*=0)
0	1	1	0	1	L	NL	Conditional on ZF=0 and SF=1 (*Not* condition ZF≠0 or SF≠1)
0	1	1	1	0			Reserved
0	1	1	1	1			Reserved
1	0	0	0	0	TF1	NTF1	Conditional on TF1=1 (*Not* condition TF1=0)
1	0	0	0	1	TF2	NTF2	Conditional on TF2=1 (*Not* condition TF2=0)
1	0	0	1	0	TAG	NTAG	Conditional on TAG=1 (*Not* condition TAG=0)
1	0	0	1	1	IN1	NIN1	Conditional on IN1=1 status. (*Not* condition IN1=0)
1	0	1	0	0	IN2	NIN2	Conditional on IN2=1 status. (*Not* condition IN2=0)
1	0	1	0	1			Unconditional
1	0	1	1	0			Reserved
1	0	1	1	1			Reserved
1	1	0	0	0	XZ	XNZ	Conditional on XZF=1 (*Not* condition XZF=0)
1	1	0	0	1	XS	XNS	Conditional on XSF=1 (*Not* condition XSF=0)
1	1	0	1	0	XG	XNG	Conditional on XSF=0 and XZF=0 (*Not* condition XSF≠0 or XZF≠0)
1	1	0	1	1			Reserved
1	1	1	0	0			Reserved
1	1	1	0	1			Reserved
1	1	1	1	0			Reserved
1	1	1	1	1			Reserved

Description If *cc* condition in Table 4–48 is true, PC + 2 is pushed onto the stack and the second word operand is loaded into the PC. If the condition is false, execution defaults to a NOP. A C*cc* instruction cannot be followed by a return (RET) instruction. No restriction applies if IRET is used instead of RET.

Syntax	Alternate Syntax	Description
CA *pma16*	CNBE *pma16*	Conditional call on above (unsigned)†
CNA *pma16*	CBE *pma16*	Conditional call on not above (unsigned)†
CB *pma16*		Conditional call on below (unsigned)
CNB *pma16*		Conditional call on not below (unsigned)
CC *pma16*		Conditional call on CF = 1
CNC *pma16*		Conditional call on CF = 0
CE *pma16*		Conditional call on equal
CNE *pma16*		Conditional call on not equal
CG *pma16*	CNLE *pma16*	Conditional call on greater (signed)†
CNG *pma16*	CLE *pma16*	Conditional call on not greater (signed)†
CIN1 *pma16*		Conditional call on IN1 = 1
CNIN1 *pma16*		Conditional call on IN1 = 0
CIN2 *pma16*		Conditional call on IN2 = 1
CNIN2 *pma16*		Conditional call on IN2 = 0
CL *pma16*	CNGE *pma16*	Conditional call on less (signed)†
CNL *pma16*	CGE *pma16*	Conditional call on not less (signed)†
CO *pma16*		Conditional call on OF = 1
CNO *pma16*		Conditional call on OF = 0
CS *pma16*		Conditional call on SF = 1
CNS *pma16*		Conditional call on SF = 0
CTAG *pma16*		Conditional call on TAG = 1
CNTAG *pma16*		Conditional call on TAG = 0
CTF1 *pma16*		Conditional call on TF1 = 1
CNTF1 *pma16*		Conditional call on TF1 = 0
CTF2 *pma16*		Conditional call on TF2 = 1
CNTF2 *pma16*		Conditional call on TF2 = 0
CZ *pma16*		Conditional call on ZF = 1
CNZ *pma16*		Conditional call on ZF = 0
CRA *pma16*	CRNBE *pma16*	Conditional call on R*x* above (unsigned)†
CRNA *pma16*	CRBE *pma16*	Conditional call on R*x* not above (unsigned)†

Syntax	Alternate Syntax	Description
CRC *pma16*		Conditional call on RCF = 1
CRNC *pma16*		Conditional call on RCF = 0
CRE *pma16*	CRZ *pma16*	Conditional call on RZF = 1 (equal)†
CRNE *pma16*	CRNZ *pma16*	Conditional call on RZF = 0 (not equal)†
CXG *pma16*	CXNLE *pma16*	Conditional call on transfer greater (signed)†
CXNG *pma16*	CXLE *pma16*	Conditional call on transfer not greater (signed)†
CXS *pma16*		Conditional call on XSF = 1
CXNS *pma16*		Conditional call on XSF = 0

† Alternate mnemonics are provided as a way of improving source code readability. They generate the same opcode as the original mnemonic. For example, CA (call above) tests the same conditions as CNBE (call not below or equal) but may have more meaning in a specific section of code.

See Also CALL, VCALL, RET, IRET

Example 4.14.9.1 `CZ 0x2010`
Call routine at program memory address 0x2010 if a previous operation has set the ZF=1 flag in STAT.

Example 4.14.9.2 `CTF1 0x2010`
Call routine at program memory address 0x2010 if a previous operation has set the TF1=1 flag in STAT.

Example 4.14.9.3 `CRNBE 0x2010`
Call routine at program memory address 0x2010 if a previous operation has set the flags RCF=1, RZF=0 in STAT.

4.14.10 CMP Compare Two Words

[label]	***name***	*src, src1* [*, mod*]	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	CMP	A*n*, {*adrs*}	Table 4–46		Table 4–46	1b
	CMP	A*n*[**~**], *imm16* [, *next A*]	2	2	N/R	2b
	CMP CMP	A*n*, A*n*~ [, *next A*] A*n*~, A*n* [, *next A*]	1	1	n_R+3	3
	CMP†	R*x*, *imm16*	2	2	N/R	4c
	CMP†	R*x*, R5	1	1	n_R+3	4d

† Does not modify A*n* status

Execution
[premodify AP if *mod* specified]
STAT flags set by *src* – *src1* operation
PC = PC + *w*

Flags Affected

src is A*n*:	OF, SF, ZF, CF are set accordingly
src is R*x*:	RCF, RZF are set accordingly
src is *{adrs}*:	TAG bit is set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CMP A*n*, {*adrs*}	0	1	0	1	1	0	0	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
CMP A*n*[**~**], *imm16* [, *next A*]	1	1	1	0	0	*next A*		**A***n*		0	1	1	0	0	1	A~	~A
	x	*imm16*															
CMP A*n*, A*n*~ [, *next A*]	1	1	1	0	0	*next A*		**A***n*		1	0	0	0	0	0	0	0
CMP A*n*~, A*n* [, *next A*]	1	1	1	0	0	*next A*		**A***n*		1	0	0	0	0	0	1	0
CMP R*x*, *imm16*	1	1	1	1	1	1	1	0	0	0	1	1	R*x*			0	0
	x	*imm16*															
CMP R*x*, R5	1	1	1	1	1	1	1	0	0	1	1	1	R*x*			0	0

Description
Subtract value of *src1* from *src* (i.e., *src*–*src1*) and only modify the status flag. Premodification of accumulator pointer is allowed with some operand types.

See Also
CMPB, CMPS, J*cc*, C*cc*

Example 4.14.10.1 `CMP A0, *R0`
Compare value at accumulator A0 and the content of data memory location pointed by R0 and change the STAT flags accordingly.

Example 4.14.10.2 `CMP A0~, 0x1400, --A`
Predecrement accumulator pointer AP0. Compare value at accumulator A0**~** to immediate value at 0x1400 and change the STAT flags accordingly.

Example 4.14.10.3 `CMP R2, 0xfe20`
Compare value at R2 to immediate value 0xfe20 and change the STAT flags accordingly.

Example 4.14.10.4 `CMP R0, R5`
Compare value at R0 to R5 and change the STAT flags accordingly.

4.14.11 CMPB Compare Two Bytes

Syntax

[label]	***name***	*src, src1*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	CMPB	A*n*, *imm8*	1	1	N/R	2a
	CMPB	R*x*, *imm8*	1	1	N/R	4b

Execution status flags set by *src* – *src1* byte
PC ⇐ PC + 1

Flags Affected *src* is A*n*: OF, SF, ZF, CF are set accordingly
src is R*x*: RCF, RZF are set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CMPB A*n*, *imm8*	1	0	1	0	0	1	1	A*n*		*imm8*							
CMPB R*x*, *imm8*	1	0	1	1	1	1	*k4*	*k3*	*k2*	*k7*	*k6*	*k5*	R*x*			*k1*	*k0*

Description Subtract value of *src1* (zero filled in upper 8 bits) from *src* (i.e., *src*–*src1*) and only modify the status flags. Contents of *src* not changed.

See Also CMP, CMPS, J*cc*, C*cc*

Example 4.14.11.1
```
CMPB A0, 0xf3
```
Compare immediate value 0xf3 to accumulator A0.

Example 4.14.11.2
```
CMPB R3, 0x21
```
Compare immediate value 0x21 to R3.

4.14.12 CMPS Compare Two Strings

Syntax

[*label*]	***name***	*src, src1*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	CMPS	A*n*, {*adrs*}	Table 4–46		Table 4–46	1b
	CMPS	A*n*[~], *pma16*	n_S+4	2	N/R	2b
	CMPS CMPS	A*n*, A*n*~ A*n*~, A*n*	n_S+3	1	n_R+3	3

Execution status flags set by (*src* – *src1*) string
PC ⇐ **PC** + *w*

Flags Affected

src is **A***n*:	OF, SF, ZF, CF are set accordingly
src1 is {*adrs*}:	TAG bit is set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CMPB A*n*, *{adrs}*	0	1	0	1	1	0	1	A*n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
CMPS A*n*[~], *pma16*	1	1	1	0	0	1	1	A*n*		0	1	1	0	0	1	**A~**	0
	x	*pma16*															
CMPS A*n*, A*n*~	1	1	1	0	0	1	1	A*n*		1	0	0	0	0	0	0	0
CMPS A*n*~, A*n*	1	1	1	0	0	1	1	A*n*		1	0	0	0	0	0	1	0

Description Subtract *src1* string from *src* string and only modify the status flags. Content of accumulators are not changed.

See Also CMPB, CMP, J*cc*, C*cc*

Example 4.14.12.1 `CMPS A0, *R0`
Compare string at data memory location pointed by R0 to A0 and change the STAT flags accordingly.

Example 4.14.12.2 `CMPS A1~, 0x1400`
Compare string at program memory location 0x1400 to A1**~** and change the STAT flags accordingly.

Example 4.14.12.3 `CMPS A2, A2~`
Compare accumulator string A2 to accumulator string A2**~** and change the STAT flags accordingly.

4.14.13 COR Correlation Filter Function

Syntax

[*label*]	***name***	*dest, src*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	COR	A*n*, *R*x*	3	1	$3(n_R+2)$	9a

Execution

With RPT N–2:
(*mask interrupts*)
RPT counter = N–2
MR = *h*[0] = first filter coefficient
x = sample data pointed by $\mathbf{R}x_{even}$
h[1] = second filter coefficient pointed by $\mathbf{R}x_{even}$+1
y = result stored in three consecutive accumulators (48 bit) pointed by **A***n*

{between every accumulation}
IF **TAG** = 1
    $\mathbf{R}x_{even}$ = $\mathbf{R}x_{even}$ + **R5** {for circular buffering}
ELSE
    $\mathbf{R}x_{even}$++ { if **R***x***++** is specified in the instruction}
ENDIF
PC ⇐ **PC** + 1
{final result}

$$y = \sum_{k=0..N-1} h[k] \cdot x[N-1-k]$$

(Execution is detailed in section 4.11)

Flags Affected **none**

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
COR A*n*, *R*x*	1	1	1	0	1	0	0	**A***n*		1	1	0	**R***x*			1	1

Description When used with repeat will execute 16 × 16 multiplication between two indirectly addressed data memory buffers, 48-bit accumulation, and a circular buffer operation. Each tap takes 3 instruction cycles. The selected register R*x* must be even. This instruction also uses R(*x*+1). This instruction must be used with RPT instruction. See section 4.11 for more detail on the setup of coefficents and sample data. During COR execution, interrupts are queued.

See Also RPT, CORK, FIR, FIRK

Example 4.14.13.1

```
RPT 0
COR A0, *R0
```

Computes the calculation for 2 tap correlation filter with 48 bit accumulation. See section 4.11 for more detail on the setup of coefficents and sample data.

4.14.14 CORK Correlation Filter Function

Syntax

[label]	***name***	*dest, src*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	CORK	A*n*, *R*x*	3	1	$3(n_R+2)$	9a

Execution

With RPT N–2:
(*mask interrupts*)
RPT counter = N–2
MR = *h*[0] = first filter coefficient
x = sample data pointed at by Rx_{even}
h[1] = second filter coefficient pointed by DP
y = result stored in three consecutive accumulators (48 bit) pointed by A*n*

{between every accumulation}
IF **TAG** = 1
    $\mathbf{R}x_{even} = \mathbf{R}x_{even} + \mathbf{R5}$ {for circular buffering}
ELSE
    $\mathbf{R}x_{even}$++ { if **R***x***++** is specified in the instruction}
ENDIF **PC** ⇐ **PC** + 1
{final result}

$$y = \sum_{k=0..N-1} h[k] \cdot x[N-1-k]$$

(Execution is detailed in section 4.11)

Flags Affected None

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CORK A*n*, *R*x*	1	1	1	0	1	0	0	A*n*		1	0	0	R*x*			1	1

Description When used with repeat will execute 16 × 16 multiplication between data memory and program memory, 48-bit accumulation, and a circular buffer operation. Each tap takes 3 instruction cycles. Selected register R*x* must be even. This instruction also uses R(*x*+1). This instruction must be used with RPT instruction. See Section 4.11 for more detail on the setup of coefficents and sample data. During CORK execution, interrupt is queued.

See Also RPT, COR, FIR, FIRK

Example 4.14.13.1

```
RPT 0
CORK A0, *R0
```

Computes the calculation for 2 tap correlation filter with 48 bit accumulation. See section 4.11 for more detail on the setup of coefficents and sample data.

4.14.15 ENDLOOP End Loop

Syntax

[label]	***name***	#	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	ENDLOOP	[*n*]	1	1	N/R	9d

Execution

If (**R4** ≥ 0)
  decrement **R4** by *n* (1 or 2)
  PC ⇐ first address after BEGLOOP
else
  NOP
  PC ⇐ PC + 1

Flags Affected None

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ENDLOOP *n*	1	1	1	1	1	1	1	1	0	0	0	0	1	*0*	0	0	*n*

Description This instruction marks the end of a loop defined by **BEGLOOP**. If register **R4** is not negative, **R4** is decremented by *n* and the loop is executed again beginning with the first instruction after the **BEGLOOP**. If **R4** is negative, a NOP instruction is executed and program exits the loop. Interrupts (queued by **BEGLOOP**) are processed according to their priority. This instruction results in an overhead of one instruction cycle per loop cycle compared to two instruction cycle if branching is used. If ENDLOOP is used without any argument, it assumes *n*=1.

See Also **BEGLOOP, INTE**

Example 4.14.15.1 See Example 4.14.7.1 in **BEGLOOP**.

4.14.16 EXTSGN Sign Extend Word

Syntax

[label]	***name***	*dest* [*, mod*]	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	EXTSGN	A*n*[~] [**,** *next A*]	1	1	n_R+3	3

Execution

[premodify AP if *mod* specified]
new most significant word of *dest* ⇐ STAT**.**SF
PC ⇐ PC + 1

Flags Affected None

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXTSGN A*n*[**~**] [**,** *next A*]	1	1	1	0	0	*next A*		**A***n*		0	1	1	1	1	0	0	**~A**

Description Copy accumulator sign flag (SF) to all 16 bits of A*n*[**~**].

See Also EXTSGNS

Example 4.14.16.1 `EXTSGN A0~, ++A`
Preincrement accumulator pointer AP0. Sign extend the accumulator A0~.

4.14.17 EXTSGNS Sign Extend String

Syntax

[label]	***name***	*dest*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	EXTSGNS	A*n*[**~**]	n_R+3	1	n_R+3	3

Execution

new most significant word of *dest* ⇐ STAT**.**SF
PC ⇐ PC + 1

Flags Affected

None

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXTSGNS A*n*[**~**]	1	1	1	0	0	1	1	A*n*		0	1	1	1	1	0	0	A~

Description

Extend the sign bit (SF) of most significant word an additional 16 bits to the left. The accumulator address is preincremented (internally) causing the sign of the addressed accumulator to be extended into the next accumulator address.

This instruction ignores the string count, executing only once, but maintains the CF and ZF status of the previous multiply or shift operation as if the sequence was a single string.

IMPORTANT:
At this stage of documentation, a bug in this instruction causes the processor to stall when an attempt is made to sign extend a string that has all zeros in it. Also, the same interrupt problem on the accumulator pointers exists if the instruction just before is not a string instruction. For customers who need the **EXTSGNS** function now as it was originally intended for string data, there is a workaround. Unfortunately, it involves the use of two accumulator pointers, the second pointing to the position in the accumulator register file that would correspond to the extended word location. For example, if a string exists in memory with the value Ox943500000000 (3 word string) and the value was to be moved to a accumulator as a 64 bit sign extended value, the following code would have been (without bugs):

```
MOV AP0, 0
MOVS A0, *R0    ; R0 POINTS TO VALUE IN MEMORY
EXTSGNS A0      ; EXTENDS THE SIGN OF ABOVE ADD IN ACC(3)
```

Since the bug causes the above function to fail, the status of the 2 least significant words is equal to zero. However, the same case will be correctly executed with the desired result with the existing bug:

```
MOV AP0, 0      ; POINT TO LSW OF ACCUM STRING
```

```
MOV AP1, 3      ; Point to loc corresponding to
                ; extended word in acc
MOVS A0, *R0    ; R0 POINTS TO VALUE IN MEMORY
EXTSGN A1       ; not string version as above
```

Alternatively, the following code can do the same thing but requires more code:

```
MOV AP0, 0   ; POINT TO LSW OF ACCUM STRING
MOV AP1, 3   ; Point to loc corresponding to
             ; extended word in acc
ZAC A1       ; INITIALIZE EXTENDED SIGN VALUE as positive
MOVS A0, *R0 ; R0 POINTS TO VALUE IN MEMORY
JNS POSITIVE ; branch around negative extension,
             ; accepting default pos extension
NOT A1       ; INVERT EXTENDED SIGN WORD FOR NEG CASE POSITIVE
    ......
```

See Also EXTSGN

Example 4.14.17.1 `EXTSGNS A0~`
Sign extend accumulator string **A0~**. See the previous italic text on the bug in this instruction at the present time.

4.14.18 FIR FIR Filter Function (Coefficients in RAM)

Syntax

[label]	***name***	*dest, src*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	FIR	A*n*, *R*x*	2	1	$2(n_R+2)$	9a

Execution

With RPT N–2:
(*mask interrupts*)
RPT counter = N–2
MR = *h*[0] = first filter coefficient
x = sample data pointed at by Rx_{even}
h[1] = second filter coefficient pointed at Rx_{even}+1
y = result stored in three consecutive accumulators (32 bit) pointed by A*n*

{between every accumulation}
IF TAG = 1
Rx_{even} = **R**x_{even} + **R5** {for circular buffering}
ELSE
Rx_{even}++ { if **R***x***++** is specified in the instruction}
ENDIF
PC ⇐ **PC** + 1
{final result}

$$y = \sum_{k=0..N-1} h[k] \cdot x[N-1-k]$$

(Execution is detailed in section 4.11)

Flags Affected None

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIR A*n*, *R*x*	1	1	1	0	1	0	0	A*n*		0	1	0	R*x*			1	1

Description

Finite impulse response (FIR) filter. Execute finite impulse response filter taps using coefficients from data memory and samples from data memory. The instruction specifies two registers, R*x* and R(*x*+1) which sequentially address coefficients and the sample buffer in the two instruction FIR tap sequence.

This instruction must be used with RPT instruction. When used with the repeat counter it will execute a 16×16 multiplication between two indirect addressed data memory buffers, 32-bit accumulation, and circular buffer operation. Executes in 2 instruction cycles.

Selected register R*x* must be even. This instruction also uses R(*x*+1). See section 4.11 for more detail on the setup of coefficients and sample data. During FIR execution, interrupt is queued.

See Also RPT, FIRK, COR, CORK

Example 4.14.18.1

```
RPT 0
FIR A0, *R0
```

Computes the calculation for 2 tap FIR filter with 32-bit accumulation. See section 4.11 for more detail on the setup of coefficients and sample data.

4.14.19 FIRK FIR Filter Function (Coefficients in ROM)

Syntax

[label]	***name***	*dest, src*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	FIRK	A*n*, *R*x*	2	1	$2(n_R+2)$	9a

Execution

With RPT N–2:
(*mask interrupts*)
RPT counter = N–2
MR = *h*[0] = first filter coefficient
x = sample data pointed by **R***x*$_{even}$
h[1] = second filter coefficient pointed by **DP**
y = result stored in three consecutive accumulators (32 bit) pointed by **A***n*
[between every accumulation}
IF **TAG** = 1
R*x*$_{even}$ = **R***x*$_{even}$ + **R5** {for circular buffering}
ELSE
R*x*$_{even}$++ { if **R***x***++** is specified in the instruction}
ENDIF
PC ⇐ **PC** + 1
{final result}

$$y = \sum_{k=0..N-1} h[k] \cdot x[N-1-k]$$

(Execution is detailed in section 4.11)

Flags Affected None

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIRK A*n*, *R*x*	1	1	1	0	1	0	0	A*n*		0	0	0	R*x*			1	1

Description

Finite impluse response (**FIR**) filter. Execute finite impulse response filter taps using coefficients from program memory and samples from data memory. Address reference for data memory is indirect using specified **R***x* and address reference for program memory is contained in DP register.

This instruction must be used with **RPT** instruction. When used with the repeat counter it will execute 16 × 16 multiplication between indirect addressed data memory buffer and program memory (coef), 32-bit accumulation, and circular buffer operation. Each tap executes in 2 cycles. See section 4.11 for more detail on the setup of coefficents and sample data. Selected register **R***x* must be even. During **FIRK** execution, interrupts are queued.

See Also **RPT, FIR, COR, CORK**

Example 4.14.19.1

```
RPT 0
FIRK A0, *R0
```

Computes the calculation for 2 tap FIR filter with 32 bit accumulation. See section 4.11 for more detail on the setup of coefficients and sample data.

4.14.20 IDLE Halt Processor

Syntax

[label]	***name***	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	IDLE	1	1	N/R	9d

Execution Stop processor clocks
PC ⇐ **PC** + 1

Flags Affected None

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDLE	1	1	1	1	1	1	1	1	0	0	0	1	0	0	0	0	0

Description Halts execution of processor. An external interrupt wakes the processor. This instruction is the only instruction to enter one of the three low power modes defined in section 2.11. Low power modes depend on the state of **ClkSpdCtrl** register bit 8 through bit 10 and the **ARM** bit in **IntGenCtrl** register.

Example 4.14.20.1

```
MOV A0, 0
OUT 0x34, A0      ; Turn off DAC
MOV A0, 0x0400    ; Turn off clock, idle bit = 1
OUT 0x3d, A0      ; Write in ClkSpdCtrl (write only)
IN A0, 0x38       ; Read IntGenCtrl register value
OR A0, A0, 0x4000 ; Set ARM = 1
OUT 0x38, A0      ; Write to IntGenCtrl
IDLE              ; Go to deep sleep mode
```

To understand this routine, refer to the *Reduced Power Modes* table in section 2.11. The bits to be set up to switch to deep sleep mode are as follows: set bits 10 of **ClkSpdCtrl** (IO address 0x3d) register to 1 and reset bits 8 and 9 of **ClkSpdCtrl** register to 0 (The **PLLM** bits are reset to zero in this example which is not a necessary operation). Note that the **ClkSpdCtrl** register is write only. Set the **ARM** bit in the **IntGenCtrl** (I/O address 0x38) register to 1 (program line 2 and 3 above). The last line executes the **IDLE** instruction which switches the processor to deep sleep mode.

4.14.21 IN Input From Port Into Word

Syntax

[label]	***name***	*dest, src1*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	IN	{*adrs*}, port4	Table 4–46		Table 4–46	6a
	IN	**A***n*[**~**]**,** *port6*	1	1	n_R+3	6b

Execution
dest ⇐ content of *port6* or *port4*
PC ⇐ **PC** + *w*

Flags Affected
dest is **A***n*: **OF**, **SF**, **ZF**, **CF** are set accordingly
dest is {*adrs}* XZF, XSF are set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IN {adrs}, port4	1	1	0	0	0	*port4*				*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
IN **A***n*[**~**]**,** *port6*	1	1	1	0	1	1	0	**A***n*		*port6*							**~A**

Description Input from I/O port. Words can be input to memory from one of 16 port addresses or one of 48 port addresses. The *port4* address is multiplied by 4 to get the actual port address.

See Also **INS, OUT, OUTS**

Example 4.14.21.1 `IN *R0, 0x0c`
Input data from port address 0x0c * 4 = 0x30 to data memory location pointed by **R0**.

Example 4.14.21.2 `IN A2~, 0x3d`
Input data from port address 0x3d to accumulator **A2~**.

4.14.22 INS Input From Port Into String

Syntax

[label]	***name***	*src, src1*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	INS	**A***n*[**~**]**,** *port6*	n_S+2	1	n_R+2	6b

Execution *dest* ⇐ content of *port6*
PC ⇐ **PC** + 1

Flags Affected *dest* is **A***n:* **OF**, **SF**, **ZF**, **CF** are set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
INS A*n*[~]**,** *port6*	1	1	1	0	1	1	1	**A***n*		*port6*						0	**~A**

Description Input string from same port, *port6*, to accumulator string. Strings can be input to accumulators from one of 64 port addresses. In this instruction, *port6* is sampled n_S+2 times. The first sample is stored in the lowest order accumulator of the string and the last sample is stored in the highest order accumulator of the string.

See Also **IN, OUT, OUTS**

Example 4.14.22.1 `INS A2, 0`
Input string starting from port 0 to accumulator string.

4.14.23 INTD Interrupt Disable

Syntax

[label]	***name***	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	INTD	1	1	N/R	9d

Execution **STAT.IM** ⇐ 0 (**IM** is **STAT** bit 4)
PC ⇐ **PC** + 1

Flags Affected None

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
INTD	1	1	1	1	1	1	1	1	0	1	0	0	1	0	0	0	0

Description Disables interrupts. Resets bit 4 (the **IM**, interrupt mask bit) of status register (**STAT**) to 0.

See Also **INTE, IRET**

Example 4.14.23.1 `INTD`
Disable interrupts. INTD must be always be immediately followed by a NOP. Any maskable interrupt occurring after the INTD – NOP sequence will not be serviced.

4.14.24 INTE Interrupt Enable

Syntax

[label]	***name***	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	INTE	1	1	N/R	9d

Execution **STAT**.**IM** ⇐ 1 (**IM** is **STAT** bit 4)
PC ⇐ **PC** + 1

Flags Affected None

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
INTE	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0

Description Enables interrupts. Sets bit 4 (the **IM**, interrupt mask bit) of status register (**STAT**) to 1.

See Also **INTD, IRET**

Example 4.1 `INTE`
Enables interrupts. Any maskable interrupts occurring after this instruction is serviced.

4.14.25 IRET Return From Interrupt

Syntax

[label]	***name***	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	IRET	2	1	N/R	5

Execution

PC ⇐ **TOS**
R7 ⇐ **R7** – 2
TOS ⇐ ***R7**

Flags Affected None

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IRET	1	1	0	1	1	1	1	0	1	0	1	1	1	1	1	1	0

See Also **RET, CALL, C*cc*, INTE, INTD**

Description Return from interrupt. Pop top of stack to program counter.

Example 4.1

```
IRET
```

Return from interrupt service routine. If used in a called subroutine, return from subroutine.

4.14.26 J*cc* Conditional Jumps

Syntax

[label]	***name***	*pma16* [, *Rmod*]	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	J*cc*	*pma16* [, *Rmod*]	2	2	N/R	7b

If true			*If **N**ot true*		
[label]	**JZ**	*pma16* [**,** *Rmod*]	*[label]*	**JNZ**	*pma16* [**,** *Rmod*]
[label]	**JS**	*pma16* [**,** *Rmod*]	*[label]*	**JNS**	*pma16* [**,** *Rmod*]
[label]	**JC**	*pma16* [**,** *Rmod*]	*[label]*	**JC**	*pma16* [**,** *Rmod*]
[label]	**JG**	*pma16* [**,** *Rmod*]	*[label]*	**JNG**	*pma16* [**,** *Rmod*]
[label]	**JE**	*pma16* [**,** *Rmod*]	*[label]*	**JNE**	*pma16* [**,** *Rmod*]
[label]	**JA**	*pma16* [**,** *Rmod*]	*[label]*	**JNA**	*pma16* [**,** *Rmod*]
[label]	**JB**	*pma16* [**,** *Rmod*]	*[label]*	**JNB**	*pma16* [**,** *Rmod*]
[label]	**JO**	*pma16* [**,** *Rmod*]	*[label]*	**JNO**	*pma16* [**,** *Rmod*]
[label]	**JRC**	*pma16* [**,** *Rmod*]	*[label]*	**JRNC**	*pma16* [**,** *Rmod*]
[label]	**JRE**	*pma16* [**,** *Rmod*]	*[label]*	**JRNE**	*pma16* [**,** *Rmod*]
[label]	**JL**	*pma16* [**,** *Rmod*]	*[label]*	**JNL**	*pma16* [**,** *Rmod*]
[label]	**JTF1**	*pma16* [**,** *Rmod*]	*[label]*	**JNTF1**	*pma16* [**,** *Rmod*]
[label]	**JTF2**	*pma16* [**,** *Rmod*]	*[label]*	**JNTF2**	*pma16* [**,** *Rmod*]
[label]	**JTAG**	*pma16* [**,** *Rmod*]	*[label]*	**JNTAG**	*pma16* [**,** *Rmod*]
[label]	**JIN1**	*pma16* [**,** *Rmod*]	*[label]*	**JNIN1**	*pma16* [**,** *Rmod*]
[label]	**JIN2**	*pma16* [**,** *Rmod*]	*[label]*	**JNIN2**	*pma16* [**,** *Rmod*]
[label]	**JXZ**	*pma16* [**,** *Rmod*]	*[label]*	**JXNZ**	*pma16* [**,** *Rmod*]
[label]	**JXS**	*pma16* [**,** *Rmod*]	*[label]*	**JXNS**	*pma16* [**,** *Rmod*]
[label]	**JXG**	*pma16* [**,** *Rmod*]	*[label]*	**JXNG**	*pma16* [**,** *Rmod*]
[label]	**JRA**	*pma16* [**,** *Rmod*]	*[label]*	**JRNA**	*pma16* [**,** *Rmod*]
[label]	**JRZP**	*pma16* [**,** *Rmod*]	*[label]*	**JRNZP**	*pma16* [**,** *Rmod*]
[label]	**JRLZP**	*pma16* [**,** *Rmod*]	*[label]*	**JRNLZP**	*pma16* [**,** *Rmod*]

Rmod	**R***x***++**
	R*x***––**
	R*x***++R5**

Execution

IF (condition = true OR unconditional)
    PC ⇐ *pma16*
ELSE
    NOP
    PC ⇐ **PC** + 2
[if post modification specified]
IF (*Rmod* = **R***x*++)
    R*x* = **R***x* + 2
ELSE IF (*Rmod* = **R***x*—)
    R*x* = **R***x* – 2
ELSE IF (*Rmod* = **R***x*++R5)
    R*x* = **R***x* **+R5**

Flags Affected **RCF** and **RZF** affected by post-modification of **R***x*.

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
J*cc* *pma16*	1	0	0	0	0	0	*Not*	*cc*					0	0	0	0	0
	x	*pma16*															
J*cc* *pma16*, **R*x*++**	1	0	0	0	0	0	*Not*	*cc*					**R*x***			0	1
	x	*pma16*															
J*cc* *pma16*, **R*x*––**	1	0	0	0	0	0	*Not*	*cc*					**R*x***			1	0
	x	*pma16*															
J*cc* *pma16*, **R*x*++R5**	1	0	0	0	0	0	*Not*	*cc*					**R*x***			1	1
	x	*pma16*															

cc					***cc* names**		**Description** **True condition (*Not* true condition)**
					cc name	**Not cc name**	
0	0	0	0	0	**Z**	**NZ**	Conditional on **ZF**=1 (*Not* condition **ZF**=0)
0	0	0	0	1	**S**	**NS**	Conditional on **SF**=1 (*Not* condition **SF**=0)
0	0	0	1	0	**C**	**NC**	Conditional on **CF**=1 (*Not* condition **CF**=0)
0	0	0	1	1	**B**	**NB**	Conditional on **ZF**=0 and **CF**=0 (*Not* condition **ZF**≠0 or **CF**≠0)
0	0	1	0	0	**A**	**NA**	Conditional on **ZF**=0 and **CF**=1 (*Not* condition **ZF**≠0 or **CF**≠1)
0	0	1	0	1	**G**	**NG**	Conditional on **SF**=0 and **ZF**=0 (*Not* condition **SF**≠0 or **ZF**≠0)
0	0	1	1	0	**E**	**NE**	Conditional if **ZF**=1 and **OF**=0 (*Not* condition **ZF**≠1 or **OF**≠0)
0	0	1	1	1	**O**	**NO**	Conditional if **OF**=1 (*Not* condition **OF**=0)
0	1	0	0	0	**RC**	**RNC**	Conditional on **RCF**=1 (*Not* condition **RCF**=0)
0	1	0	0	1	**RA**	**RNA**	Conditional on **RZF**=0 and **RCF**=1 (*Not* condition **RZF**≠0 or **RCF**≠1)
0	1	0	1	0	**RE**	**RNE**	Conditional on **RZF**=1 (*Not* condition **RZF**=0)
0	1	0	1	1	**RZP**	**RNZP**	Conditional on value of **R*x***=0 (*Not* condition **R*x***≠0)
0	1	1	0	0	**RLZP**	**RNLZP**	Conditional on MSB of **R**x=1. (*Not* condition MSB of **R*x***=0)
0	1	1	0	1	**L**	**NL**	Conditional on **ZF**=0 and **SF**=1 (*Not* condition **ZF**≠0 or **SF**≠1)
0	1	1	1	0			reserved
0	1	1	1	1			reserved
1	0	0	0	0	**TF1**	**NTF1**	Conditional on **TF1**=1 (*Not* condition **TF1**=0)
1	0	0	0	1	**TF2**	**NTF2**	Conditional on **TF2**=1 (*Not* condition **TF2**=0)
1	0	0	1	0	**TAG**	**NTAG**	Conditional on **TAG**=1 (*Not* condition **TAG**=0)
1	0	0	1	1	**IN1**	**NIN1**	Conditional on **IN1**=1 status. (*Not* condition **IN1**=0)
1	0	1	0	0	**IN2**	**NIN2**	Conditional on **IN2**=1 status. (*Not* condition **IN2**=0)
1	0	1	0	1			Unconditional
1	0	1	1	0			reserved
1	0	1	1	1			reserved
1	1	0	0	0	**XZ**	**XNZ**	Conditional on **XZF**=1 (*Not* condition **XZF**=0)
1	1	0	0	1	**XS**	**XNS**	Conditional on **XSF**=1 (*Not* condition **XSF**=0)
1	1	0	1	0	**XG**	**XNG**	Conditional on **XSF**=0 and **XZF**=0 (*Not* condition **XSF**≠0 or **XZF**≠0)
1	1	0	1	1			reserved

cc					***cc* names**		**Description** **True condition (*Not* true condition)**
					cc name	**Not cc name**	
1	1	1	0	0			reserved
1	1	1	0	1			reserved
1	1	1	1	0			reserved
1	1	1	1	1			reserved

Description **PC** is replaced with second word operand if condition is true (or unconditional). If test condition is false, a **NOP** is executed.

Syntax	**Alternate Instruction**	**Description**
JA *pma16* [, *Rmod*]	**JNBE**	Conditional jump on above (unsigned)
JNA *pma16* [, *Rmod*]	**JBE**	Conditional jump on not above (unsigned)
JB *pma16* [, *Rmod*]	**JNAE**	Conditional jump on below (unsigned)
JNB *pma16* [, *Rmod*]	**JAE**	Conditional jump on not below (unsigned)
JC *pma16* [, *Rmod*]		Conditional jump on **CF** = 1
JNC *pma16* [, *Rmod*]		Conditional jump on **CF** = 0
JE *pma16* [, *Rmod*]		Conditional jump on equal
JNE *pma16* [, *Rmod*]		Conditional jump on not equal
JG *pma16* [, *Rmod*]	**JNLE**	Conditional jump on greater (signed)
JNG *pma16* [, *Rmod*]	**JLE**	Conditional jump on not greater (signed)
JIN1 *pma16* [, *Rmod*]		Conditional jump on port D pin PD_0=1
JNIN1 *pma16* [, *Rmod*]		Conditional jump on port D pin PD_0=0
JIN2 *pma16* [, *Rmod*]		Conditional jump on port D pin PD_1=1
JNIN2 *pma16* [, *Rmod*]		Conditional jump on port D pin PD_1=0
JL *pma16* [, *Rmod*]	**JNGE**	Conditional jump on less than(signed)
JNL *pma16* [, *Rmod*]	**JGE**	Conditional jump on not less than (signed)
JO *pma16* [, *Rmod*]		Conditional jump on **OF** = 1
JNO *pma16* [, *Rmod*]		Conditional jump on **OF** = 0
JRA *pma16* [, *Rmod*]		Conditional jump on **R***x* above (unsigned)
JRNA *pma16* [, *Rmod*]	**JRBE**	Conditional jump on **R***x* not above (unsigned)
JRC *pma16* [, *Rmod*]		Conditional jump on **XCF** = 1
JRNC *pma16* [, *Rmod*]		Conditional jump on **XCF** = 0
JRE *pma16* [, *Rmod*]	**JRZ**	Conditional jump on **XZF** = 1 (equal)†
JRNE *pma16* [, *Rmod*]	**JRNZ**	Conditional jump on **XZF** = 0 (not equal)†
JRNBE *pma16* [, *Rmod*]		Conditional jump on **R***x* not below or equal (unsigned)†
JRLZP *pma16* [, *Rmod*]		Conditional jump on **R***x* < 0 after post-mod

Syntax	Alternate Instruction	Description
JRNLZP *pma16* [, *Rmod*]		Conditional jump on **R***x* ≥ 0 after post-mod
JRZP *pma16* [, *Rmod*]		Conditional jump on **R***x* = 0 after post-mod
JRNZP *pma16* [, *Rmod*]		Conditional jump on **R***x* ≠ 0 after post-mod
JS *pma16* [, *Rmod*]		Conditional jump on **SF** = 1
JNS *pma16* [, *Rmod*]		Conditional jump on **SF** = 0
JTAG *pma16* [, *Rmod*]		Conditional jump on **TAG** = 1
JNTAG *pma16* [, *Rmod*]		Conditional jump on **TAG** = 0
JTF1 *pma16* [, *Rmod*]		Conditional jump on **TF1** = 1
JNTF1 *pma16* [, *Rmod*]		Conditional jump on **TF1** = 0
JTF2 *pma16* [, *Rmod*]		Conditional jump on **TF2** = 1
JNTF2 *pma16* [, *Rmod*]		Conditional jump on **TF2** = 0
JXG *pma16* [, *Rmod*]	**JXNLE**	Conditional jump on transfer greater (signed)†
JXNG *pma16* [, *Rmod*]	**JXLE**	Conditional jump on transfer not greater (signed)†
JXS *pma16* [, *Rmod*]		Conditional jump on transfer **SF** = 1
JXNS *pma16* [, *Rmod*]		Conditional jump on transfer **SF** = 0
JXZ *pma16* [, *Rmod*]		Conditional jump on transfer **ZF** = 1 (zero)
JXNZ *pma16* [, *Rmod*]		Conditional jump on transfer **ZF** = 0 (not equal)
JZ *pma16* [, *Rmod*]		Conditional jump on **ZF** = 1
JNZ *pma16* [, *Rmod*]		Conditional jump on **ZF** = 0

† Alternate mnemonics are provided as a way of improving source code readability. They generate the same opcode as the original mnemonic. For example, **JA** (jump above) tests the same conditions as **JNBE** (jump not below or equal) but may have more meaning in a specific section of code.

See Also **JMP, CALL, C*cc***

Example 4.14.27.1 `JNZ 0x2010`
Jump to program memory location 0x2010 if the result is not zero.

Example 4.14.27.2 `JE 0x2010, R3++R5`
Jump to program memory location 0x2010 if flag **RZF** = 1. Increment **R3** by **R5**. Since this jump instruction does not have a P at the end, post-modification is NOT reflected in the **STAT** register. Thus, if **R3** becomes zero, **RZF** is not updated.

Example 4.14.27.3 `JIN1 0x2010, R1--`
Jump to program memory location 0x2010 if I/O port address $\mathbf{PD_0}$ pin has a value of 1. Decrement **R1** by 2.

Example 4.14.27.4 `JTAG 0x2010, R2++`
Jump to program memory location 0x2010 if **TAG** bit of **STAT** is zero. Increment **R2** by 2.

4.14.27 JMP Unconditional Jump

Syntax

[label]	***name***	*dest* [, *mod*]	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	JMP	*pma16*	2	2	N/R	7b
	JMP	*pma16*, ***R****x****++***	2	2	N/R	7b
	JMP	*pma16,* **R***x***––**	2	2	N/R	7b
	JMP	*pma16,* **R***x***++R5**	2	2	N/R	7b
	JMP	***A**n*	2	1	N/R	7b

Execution **PC** ⇐ *dest*
[Post–modify **R***x* if specified]

Flags Affected **RCF** and **RZF** affected by post–modification of **R***x*

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JMP *pma16*	1	0	0	0	0	1	0	1	0	1	0	1	0	0	0	0	0
	x	*pma16*															
JMP *pma16*, **R***x***++**	1	0	0	0	0	0	0	1	0	1	0	1	**R***x*			0	1
	x	*pma16*															
JMP *pma16*, **R***x***––**	1	0	0	0	0	0	0	1	0	1	0	1	**R***x*			1	0
	x	*pma16*															
JMP *pma16*, **R***x***++R5**	1	0	0	0	0	0	0	1	0	1	0	1	**R***x*			1	1
	x	*pma16*															
JMP *A*n*	1	0	0	0	1	0	0	**A***n*		0	0	0	0	0	0	0	0

Description

Instruction	**Operation**
JMP *pma16*[, *mod*]	**PC** is replaced with second word operand. Post modification of **R***x* register is done if specified.
JMP *A*n*	**PC** is replaced with content of accumulator **A***n*.

See Also **J*cc*, CALL, C*cc***

Example 4.14.26.1 `JMP 0x2010, R2––`
Jump unconditionally to program memory location 0x2010. Decrement **R2** by 2.

Example 4.14.26.2 `JMP *A3`
Jump unconditionally to program memory location stored in accumulator **A3**.

4.14.28 MOV Move Data Word From Source to Destination

Syntax

[label]	***name***	*dest, src,* [, *next A*]	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	MOV	{*adrs*}**, A***n*[**~**] [, *next A*]	Table 4–46		Table 4–46	1a
	MOV	**A***n*[**~**]**,** {*adrs*} [, *next A*]	Table 4–46		Table 4–46	1a
	MOV	{*adrs*}**, *A***n*	Table 4–46		Table 4–46	1b
	MOV	**A***n*[**~**]**,** *imm16* [, *next A*]	2	2	N/R	2b
	MOV	**MR,** *imm16* [, *next A*]	2	2	N/R	2b
	MOV	**A***n***, A***n***~** [, *next A*]	1	1	n_R+3	3
	MOV	**A***n*[**~**]**, PH** [, *next A*]	1	1	n_R+3	3
	MOV	**SV, A***n*[**~**] [, *next A*]	1	1	n_R+3	3
	MOV	**PH, A***n*[**~**] [, *next A*]	1	1	n_R+3	3
	MOV	**A***n*[**~**]**, *A***n*[**~**] [, *next A*]	1	1	n_R+3	3
	MOV	**MR, A***n*[**~**] [, *next A*]	1	1	n_R+3	3
	MOV	{*adrs*}**, R***x*	Table 4–46		Table 4–46	4a
	MOV	**R***x***,** {*adrs*}	Table 4–46		Table 4–46	4a
	MOV	**R***x***,** *imm16*	2	2	N/R	4c
	MOV	**R***x***, R5**	1	1	n_R+3	4d
	MOV	**SV,** $\{adrs\}_4$	1	1	n_R+3	5
	MOV	**PH,** {*adrs*}	Table 4–46		Table 4–46	5
	MOV	**MR,** {*adrs*}	Table 4–46		Table 4–46	5
	MOV	**AP***n***,** {*adrs*}	Table 4–46		Table 4–46	5
	MOV	**STAT,** {*adrs*}	Table 4–46		Table 4–46	5
	MOV	**TOS,** {*adrs*}	Table 4–46		Table 4–46	5
	MOV	{*adrs*}**, PH**	Table 4–46		Table 4–46	5
	MOV	{*adrs*}**, MR**	Table 4–46		Table 4–46	5
	MOV	{*adrs*}**, STAT**	Table 4–46		Table 4–46	5
	MOV	{*adrs*}**, STR**	Table 4–46		Table 4–46	5
	MOV	{*adrs*}**, DP**	Table 4–46		Table 4–46	5
	MOV	{*adrs*}**, SV**	Table 4–46		Table 4–46	5
	MOV	{*adrs*}**, AP***n*	Table 4–46		Table 4–46	5
	MOV	{*adrs*}**, TOS**	Table 4–46		Table 4–46	5
	MOV	**STR,** $\{adrs\}_8$	Table 4–46		Table 4–46	5
	MOV	{*flagadrs*}**, TF***n*	1	1	n_R+3	8a
	MOV	**TF***n***,** {*flagadrs*}	1	1	n_R+3	8a

[label]	***name***	*dest, src,* [, *next A*]	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	MOV	**TF***n***,** {*cc*} [**, R***x*]	1	1	N/R	8b
	MOV	**STR,** *imm8*	1	1	N/R	9b
	MOV	**SV,** *imm4*	1	1	N/R	9b
	MOV	**AP***n***,** *imm5*	1	1	N/R	9c

Execution

[premodify **AP** if *mod* specified]
dest ⇐ *src*
PC ⇐ **PC** + *w*

Flags Affected

dest is **A***n*:	**OF**, **SF**, **ZF**, **CF** are set accordingly
dest is **R***x*:	**RCF**, **RZF** are set accordingly
dest is {*adrs*}:	**XSF**, **XZF** are set accordingly
src is {*adrs*}	**TAG** bit is set accordingly
src is {*flagadrs*}	**TAG** bit is set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOV {*adrs*}, **A***n***[~]** [, *next A*]	0	0	1	1	**A~**	next A		**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV A*n***[~]**, {*adrs*} [*, next A*]	0	0	1	0	**A~**	next A		**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV {*adrs*}, ***A***n*	0	1	0	1	1	1	0	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV A*n***[~]**, *imm16* [*, next A*]	1	1	1	0	0	next A		**A***n*		0	0	1	0	0	1	0	**~A**
	x	*imm16*															
MOV MR, *imm16* [, *next A*]	1	1	1	0	0	next A		**A***n*		1	1	1	0	0	1	0	0
	x	*imm16*															
MOV A*n*, **A***n***~** [, *next A*]	1	1	1	0	0	next A		**A***n*		0	0	1	1	1	0	**A~**	**~A**
MOV A*n***[~], PH** [, *next A*]	1	1	1	0	0	next A		**A***n*		0	1	1	1	0	0	**A~**	**~A**
MOV SV, A*n***[~]** [, *next A*]	1	1	1	0	0	next A		**A***n*		1	0	1	0	0	0	**A~**	0
MOV PH, A*n***[~]** [, *next A*]	1	1	1	0	0	*next A*		**A***n*		1	0	1	0	1	0	**A~**	0
MOV A*n***[~]**, ***A***n***[~]** [, *next A*]	1	1	1	0	0	*next A*		**A***n*		0	0	0	1	0	0	**A~**	**~A**
MOV MR, A*n***[~]** [, *next A*]	1	1	1	0	0	*next A*		**A***n*		1	0	1	1	0	0	**A~**	0
MOV {*adrs*}, **R***x*	1	1	1	1	0	0	**R***x*			{*adrs*}							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV R*x*, {*adrs*}	1	1	1	1	0	1	**R***x*			{*adrs*}							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV R*x*, *imm16*	1	1	1	1	1	1	1	0	0	0	1	0	**R***x*			0	0
	x	*imm16*															

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOV R*x*, **R5**	1	1	1	1	1	1	1	0	0	1	1	0	**R***x*			0	0
MOV SV, *imm4*	1	1	1	1	1	1	0	1	0	0	0	0	0	*imm4*			
MOV SV, {*adrs*}$_4$	1	1	0	1	1	0	0	0	0	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV PH, {*adrs*}	1	1	0	1	1	0	0	0	1	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV MR, {*adrs*}	1	1	0	1	1	1	0	0	0	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV AP*n*, {*adrs*}	1	1	0	1	1	0	1	**AP***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV STAT, {*adrs*}	1	1	0	1	1	1	1	1	1	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV TOS, {*adrs*}	1	1	0	1	1	0	0	1	0	*adrs*							
	x	*dma16* (for direct) or off*set16* (long relative) [see section 4.13]															
MOV {*adrs*}, **PH**	1	1	0	1	0	0	0	0	1	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV {*adrs*}, **MR**	1	1	0	1	0	1	0	0	0	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV {*adrs*}, **STAT**	1	1	0	1	0	0	0	1	0	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV {*adrs*}, **STR**	1	1	0	1	0	0	0	1	1	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV {*adrs*}, **DP**	1	1	0	1	0	1	0	1	0	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV {adrs}, **SV**	1	1	0	1	0	0	0	0	0	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV {adrs}, **AP***n*	1	1	0	1	0	0	1	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV {*adrs*}, **TOS**	1	1	0	1	0	1	0	1	1	*adrs*							
	x	*dma16* (for direct) or offset16 (long relative) [see section 4.13]															
MOV STR, {*adrs*}$_8$	1	1	0	1	1	0	0	1	1	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV {*flagadrs*}, **TF***n*	1	0	0	1	1	*flg*	*Not*	0	0	1	*flagadrs*						
MOV TF*n*, {*flagadrs*}	1	0	0	1	1	*flg*	*Not*	0	0	0	*flagadrs*						
MOV TF*n*, {*cc*} [, **R***x*]	1	0	0	1	0	*flg*	*Not*	*cc*					**R***x*			0	0
MOV STR, *imm8*	1	1	1	1	1	1	0	0	1	*imm8*							
MOV AP*n*, *imm5*	1	1	1	1	1	0	1	**A***n*		0	0	0	*imm5*				

Description Copy value of *src* to *dest*. Premodification of accumulator pointers is allowed with some operand types.

Syntax	Description
MOV A*n*[**~**], {*adrs*} [, *next A*]	Move data memory word to **A***n*[**~**]†
MOV {*adrs*}, **A***n*[**~**] [, *next A*]	Move **A***n*[**~**] word to data memory
MOV A*n*[**~**], *imm16* [, *next A*]	Move immediate word to **A***n*[**~**]†
MOV MR, *imm16* [, *next A*]	Move immediate word to multiply register†
MOV A*n*, **A***n***~** [, *next A*]	Move **A***n***~** word to **A***n*
MOV A*n***~**, **A***n* [, *next A*]	Move **A***n* word to **A***n***~**
MOV A*n*[**~**], PH [, *next A*]	Move product high reg to **A***n*[**~**]†
MOV SV, **A***n*[**~**] [, *next A*]	Move lower 4 bits of **A***n*[**~**] to **SV** register
MOV PH, **A***n*[**~**] [, *next A*]	Move **A***n*[**~**] to **PH** register
MOV MR, **A***n*[**~**] [, *next A*]	Move **A***n*[**~**] to **MR** register in signed multiplier mode‡
MOV A*n*[**~**], ***A***n*[**~**] [, *next A*]	Move program memory word at ***A***n*[**~**] to **A***n*[**~**]†
MOV {*adrs*}, **R***x*	Move **R***x* word to data memory
MOV R*x*, {*adrs*}	Move data memory word to **R***x*
MOV R*x*, *imm16*	Move immediate word to **R***x*
MOV R*x*, **R5**	Move **R5** to **R***x*
MOV PH, {*adrs*}	Move data memory word to product high (**PH**) register
MOV MR, {*adrs*}	Move data memory word to **MR**, set multiplier signed mode‡
MOV {*adrs*}, ***A***n*	Move ROM word at ***A***n* to data memory
MOV AP*n*, {*adrs*}	Move data memory word (lower 6 bits)to **AP***n* register
MOV STAT, {*adrs*}	Move data memory word to status register (**STAT**)
MOV SV, {*adrs*}‡	Move data memory value (lower 4 bits) to shift value (**SV**) register
MOV TOS, {*adrs*}	Move data memory word to top of stack (**TOS**)
MOV {*adrs*}, **PH**	Move product high (**PH**) register to data memory
MOV {*adrs*}, **MR**	Move Multiplier register (**MR**) to data memory
MOV {*adrs*}, **STAT**	Move status register (**STAT**) to data memory
MOV {*adrs*}, **STR**	Move string register (**STR**) byte to data memory
MOV {*adrs*}, **DP**	Move data pointer (**DP**) to data memory
MOV {*adrs*}, **SV**	Move shift value (**SV**) (4 bits) to data memory
MOV {*adrs*}, **AP***n*	Move **AP***n* register to data memory
MOV STR, {*adrs*}§	Move data memory byte to string register (**STR**)
MOV {*adrs*}, **TOS**	Move top of stack (**TOS**) to data memory word
MOV TF*n*, {*flagadrs*}	Move data flag to **TF***n* in **STAT** register
MOV {*flagadrs*}, **TF***n*	Move **TF***n* from **STAT** register to memory flag†
MOV TF*n*, {*cc*} [, **R***x*]	Load logic value of test condition to **TF***n* bit in **STAT** register¶
MOV SV, *imm4*	Move immediate value to shift value (**SV**) register

Syntax	Description
MOV STR, *imm8*	Move immediate byte to String Register (**STR**)
MOV AP*n*, *imm5*	Move immediate 5-bit value to **AP***n* register

† Accumulator condition flags are modified to reflect the value loaded into either **A***n* or **A***n***~**.
‡ Signed multiplier mode resets **UM** (bit 1 in status register) to 0
¶ Load the logic value of the test condition to the **TF***n* bit in the status register (**STAT**). If the condition is true, **TF***n*=1, else **TF***n*=0.

See Also **MOVU, MOVT, MOVB, MOVBS, MOVS**

Example 4.14.28.1 `MOV A0, *0x0200 * 2, ++A`
Preincrement accumulator pointer **AP0**. Copy content of word memory location 0x0200 to accumulator **A0**.

Example 4.14.28.2 `MOV *0x0200 * 2, A0, ++A`
Preincrement accumulator pointer **AP0**. Copy content of accumulator **A0** to word memory location 0x0200.

Example 4.14.28.3 `MOV *0x0200 * 2, *A1`
Transfer content of program memory location pointed by A1 to word data memory location 0x0200.

Example 4.14.28.4 `MOV A2, 0xf200, --A`
Predecrement accumulator pointer **AP2**. Load accumulator **A2** with immediate value 0xf200.

Example 4.14.28.5 `MOV A0, A0~`
Copy content of accumulator **A0~** to accumulator **A0**.

Example 4.14.28.6 `MOV A0~, A0`
Copy content of accumulator **A0** to accumulator **A0~**.

Example 4.14.28.7 `MOV A0~, PH`
Copy content of **PH** to accumulator **A0~**.

Example 4.14.28.8 `MOV SV, A3, --A`
Predecrement accumulator pointer **AP3**. Copy content of accumulator **A3** to **SV**.

Example 4.14.28.9 `MOV PH, A3`
Copy content of accumulator **A3** to **PH**.

Example 4.14.28.10 `MOV MR, A3, --A`
Predecrement accumulator pointer **AP3**. Copy content of accumulator **A3** to **MR**.

Example 4.14.28.11 `MOV A1~, *A1`
Transfer program memory value pointed by accumulator **A1** to accumulator **A1~**. This is a table lookup instruction.

Example 4.14.28.12 `MOV *0x0200 * 2, R0`
Store content of **R0** to data memory word location 0x0200.

Example 4.14.28.13 `MOV R1, 0x0200 * 2`
Load immediate word memory address 0x0200 to **R1**.

Example 4.14.28.14 `MOV R7, (0x0280 – 32) * 2`
Load **R7** (stack register) with the starting value of stack, i.e., 0x0260.

Example 4.14.28.15 `MOV *0x0200 * 2, R0`
Store **R0** to data memory word location 0x0200.

Example 4.14.28.16 `MOV R0, R5`
Transfer **R5** to **R0**.

Example 4.14.28.17 `MOV AP2, *R3`
Copy content of data memory location stored in **R3** to accumulator pointer **AP2**.

Example 4.14.28.18 `MOV *R6 + 8 * 2, DP`
Copy data pointer (**DP**) to data memory word location pointed by **R6** offset by 8 location (short relative addressing).

Example 4.14.28.19 `MOV STR, *0x0200 * 2`
Copy the **STR** register with the content of word memory location 0x0200.

Example 4.14.28.20 `MOV *R6+0x20, TF2`
Copy **TF2** flag to the flag bit in relative flag location **R6** offset by 0x20.

Example 4.14.28.21 `MOV TF1, ZF`
Copy status of **ZF** flag in **STAT** register to **TF1**.

Example 4.14.28.22 `MOV SV, 4 – 2`
Load **SV** register with a constant value 2.

Example 4.14.28.23 `MOV AP3, 23 – 16`
Load accumulator pointer **AP3** with value 7.

4.14.29 MOVAPH Move With Adding PH

Syntax

[label]	***name***	*dest, src, src1*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	MOVAPH	**A***n***, MR,** {*adrs*}	Table 4–46		Table 4–46	1b

Execution

A*n* ⇐ **A***n* + **PH**
MR ⇐ contents of {*adrs*}
PC ⇐ **PC** + *w*

Flags Affected **TAG, OF**, **SF**, **ZF**, **CF** are set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOVAPH A*n***, MR,** {*adrs*}	0	1	1	0	1	0	0	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															

Description Move RAM word to **MR** register, add **PH** to **A***n* in parallel.

See Also **MOVAPHS, MOVTPH, MOVTPHS, MOVSPH, MOVSPHS**

Example 4.14.34.1

```
MOVAPH A0, MR, *R3+R5
```

Load the contents of the byte address created by adding **R3** and **R5** to the **MR** register. At the same time, add accumulator **A0** to the **PH** register and store the result in **A0**.

4.14.30 MOVAPHS Move With Adding PH

Syntax

[label]	***name***	*dest, src, src1*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	MOVAPHS	**A***n***, MR,** {*adrs*}	Table 4–46		Table 4–46	1b

Execution

A*n* ⇐ **A***n* + **PH**
MR ⇐ contents of {*adrs*}
PC ⇐ **PC** + *w*

Flags Affected **TAG, OF**, **SF**, **ZF**, **CF** are set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOVAPHS A*n***, MR,** {*adrs*}	0	1	1	0	1	0	1	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															

Description Move RAM word to **MR**, add **PH** to second word in **A***n* string. Certain restriction applies to the use of this instruction when interrupts are occuring on the background. See section 4.8 for more details.

See Also **MOVAPH, MOVTPH, MOVTPHS, MOVSPH, MOVSPHS**

Example 4.14.35.1

```
MOVAPHS A0, MR, *R3+R5
```

Load the content of byte address created by adding **R3** and **R5** to **MR** register. At the same time, add second word in accumulator string **A0** to **PH** register, store result in **A0** string.

4.14.31 MOVB Move Byte From Source to Destination

Syntax

[label]	***name***	*dest, src*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	MOVB	**A***n***,** {*adrs*}	Table 4–46		Table 4–46	1b
	MOVB	{*adrs*}**, A***n*	Table 4–46		Table 4–46	1b
	MOVB	**A***n***,** *imm8*	1	1	N/R	2a
	MOVB	**MR,** *imm8*	1	1	N/R	2a
	MOVB	**R***x***,** *imm8*	1	1	N/R	2b

Execution

dest ⇐ *src*
PC ⇐ **PC** + *w*

Flags Affected

dest is **A***n*:	**OF**, **SF**, **ZF**, **CF** are set accordingly
dest is **R***x*:	**RCF**, **RZF** are set accordingly
dest is {*adrs*}:	**XSF**, **XZF** are set accordingly
src is {*adrs*}	**TAG** bit is set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOVB A*n***,** {*adrs*}	0	1	0	0	1	1	0	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOVB {*adrs*}**, A***n*	0	1	0	1	0	0	0	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOVB A*n***,** *imm8*	1	0	1	0	0	0	1	**A***n*		*imm8*							
MOVB MR, *imm8*	1	0	1	0	1	1	1	**A***n*		*imm8*							
MOVB R*x***,** *imm8*	1	0	1	1	1	0	*k4*	*k3*	*k2*	*k7*	*k6*	*k5*	**R***x*			*k1*	*k0*

Description Copy value of unsigned *src* byte to *dest* byte.

Syntax	Description
MOVB A*n***,** {*adrs*}	Move data memory byte to **A***n*†
MOVB {*adrs*}, **A***n*	Move **A***n* byte to data memory
MOVB A*n***,** *imm8*	Move immediate byte to **A***n*†
MOVB MR, *imm8*	Move immediate byte to multiply register (MR)‡
MOVB R*x***,** *imm8*	Move immediate byte to **R***x*

† Zeros loaded to upper 8 bits of **A***n*.
‡ Status flags are not modified

See Also **MOVU, MOV, MOVT, MOVBS, MOVS**

Example 4.14.29.1
```
MOVB A0, *R2
```
Copy data memory byte pointed by **R2** to accumulator **A0**.

Example 4.14.29.2 `MOVB *R2, A0`
Copy lower 8 bits of accumulator **A0** to the data memory byte pointed by **R2**.

Example 4.14.29.3 `MOVB A0, 0xf2`
Load accumulator **A0** with value of 0xf2.

Example 4.14.29.4 `MOVB MR, 34`
Load **MR** register with immidiate value of 34 (decimal).

Example 4.14.29.5 `MOVB R2, 255`
Load **R2** with immidiate value of 255 (decimal).

4.14.32 MOVBS Move Byte String from Source to Destination

Syntax

[label]	***name***	*dest, src*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	MOVBS	**A***n***,** {*adrs*}$_8$	Table 4–46		Table 4–46	1b
	MOVBS	{*adrs*}**, A***n*	Table 4–46		Table 4–46	1b

Execution
dest ⇐ *src*
PC ⇐ **PC** + *w*

Flags Affected

dest is **A***n*:	**OF**, **SF**, **ZF**, **CF** are set accordingly
dest is {*adrs*}:	**XSF**, **XZF** are set accordingly
src is {*adrs*}	**TAG** bit is set to bit 17th value

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOVBS A*n***,** {*adrs*}$_8$	0	1	0	0	1	1	1	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOVBS {*adrs*}$_8$**, A***n*	0	1	0	1	0	0	0	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															

Description Copy value of *src* byte to *dest*.

Syntax	**Description**
MOVBS A*n***,** {*adrs*}	Move data memory byte string to **A***n* word string
MOVB {*adrs*}, **A***n*	Move **A***n* byte string to data memory

See Also **MOVU, MOV, MOVT, MOVB, MOVS**

Example 4.14.30.1 `MOVBS A2, *0x0200`
Transfer the byte string at data memory location 0x0200 to accumulator string **A2**.

Example 4.14.30.2 `MOVBS *0x0200, A2`
Transfer accumulator string **A2** to data memory byte string location 0x0200.

4.14.33 MOVS Move String from Source to Destination

Syntax

[label]	***name***	*dest, src*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	MOVS	**A***n*[~], {*adrs*}	Table 4–46		Table 4–46	1a
	MOVS	{*adrs*}, **A***n*[~]	Table 4–46		Table 4–46	1a
	MOVS	{*adrs*}, ***A***n*	Table 4–46		Table 4–46	1b
	MOVS	**A***n*[~], *pma16*	n_S+4	2	N/R	2b
	MOVS†	**A***n*[~], **PH**	1	1	1	3
	MOVS	**A***n*, **A***n*~	n_S+2	1	n_R+2	3
	MOVS	**A***n*[~], ***A***n*[~]	n_S+4	1	n_R+4	3

† Certain restriction applies to the use of this instruction when interrupts are occuring on the background. See Section 4.8 for more detail.

Execution

dest ⇐ *src*
PC ⇐ **PC** + *w*

Flags Affected

dest is **A***n*: **OF**, **SF**, **ZF**, **CF** are set accordingly
dest is {*adrs*}: **XSF**, **XZF** are set accordingly
src is {*adrs*} **TAG** bit is set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOVS A*n*[~], {*adrs*}	0	0	1	0	**A~**	1	1	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOVS {*adrs*}, **A***n*[~]	0	0	0	1	**A~**	1	1	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOVS {*adrs*}, ***A***n*	0	1	0	1	1	1	1	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOVS A*n*[~], *pma16*	1	1	1	0	0	1	1	**A***n*		0	0	1	0	0	1	**A~**	**~A**
	x	*pma16*															
MOVS PH, A*n*[~]	1	1	1	0	0	1	1	**A***n*		1	0	1	0	1	0	**A~**	0
MOVS SV, A*n*[~]	1	1	1	0	0	1	1	**A***n*		1	0	1	0	0	0	**A~**	0
MOVS A*n*[~], **PH**	1	1	1	0	0	1	1	**A***n*		0	0	1	0	0	0	**A~**	**~A**
MOVS A*n*, **A***n*~	1	1	1	0	0	1	1	**A***n*		0	0	1	1	1	0	**A~**	**~A**
MOVS MR, A*n*[~]	1	1	1	0	0	1	1	**A***n*		1	0	1	1	0	0	**A~**	0
MOVS A*n*[~], ***A***n*[~]	1	1	1	0	0	1	1	**A***n*		0	0	0	1	0	0	**A~**	**~A**

Description Copy value of *src* string to *dest* string. Premodification of accumulator pointers is allowed with some operand types.

Syntax	Description
MOVS A*n*[**~**], {*adrs*}	Move data memory word string to **A***n*[**~**] string
MOVS {*adrs*}, **A***n*[**~**]	Move **A***n*[**~**] string to data memory
MOVS {*adrs*}, ***A***n*	Move program memory string at ***A***n* to data memory
MOVS A*n*[**~**], *pma16*	Move program memory string to **A***n*[**~**] string
MOVS A*n*, **A***n***~**	Move **A***n***~** string to **A***n*
MOVS A*n***~**, **A***n*	Move **A***n* string to **A***n***~** string
MOVS A*n*[**~**], **PH**	Move product high reg to **A***n*[**~**], string mode. This instruction ignores the string count, executing only once but maintains the CF and ZF status of the previous multiply or shift operation as if the sequence was a single string.
MOVS A*n*[**~**], ***A***n*[**~**]	Move program memory string at ***A***n*[**~**] to **A***n*[**~**]

See Also **MOVU, MOV, MOVT, MOVB, MOVBS**

Example 4.14.31.1 `MOVS A2~, *R6`
Load the string pointed by **R6** to accumulator string **A2~**.

Example 4.14.31.2 `MOVS *R4, A2~`
Copy the accumulator string **A2~** to data memory location pointed by **R4**.

Example 4.14.31.3 `MOVS *0x0100 * 2, *A0`
Transfer the program memory word string pointed by content of **A0** to the data memory word location 0x0100. This is a lookup instruction.

Example 4.14.31.4 `MOVS A2~, 0x1400`
Transfer program memory string at 0x1400 to accumulator string **A2~**.

Example 4.14.31.5 `MOVS A1, A1~`
Transfer accumulator string **A1~** to accumulator string **A1**.

Example 4.14.31.6 `MOVS A1~, A1`
Transfer accumulator string **A1** to accumulator string **A1~**.

Example 4.14.31.7 `MOVS A2, PH`
Transfer value in **PH** to accumulator string **A2**. **PH** is copied to the second word of the string.

4.14.34 MOVSPH Move With Subtract from PH

Syntax

[label]	***name***	*dest, src, src1*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	MOVSPH	**A***n***, MR,** {*adrs*}	Table 4–46		Table 4–46	1b

Execution

A*n* ⇐ **A***n* – **PH**
MR ⇐ contents of {*adrs*}
PC ⇐ **PC** + *w*

Flags Affected **TAG, OF**, **SF**, **ZF**, **CF** are set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOVSPH A*n***, MR,** {*adrs*}	0	1	1	0	0	1	0	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															

Description Move data memory to **MR**, subtract **PH** from **A***n,* store result in **A***n*.

See Also **MOVSPHS, MOVAPH, MOVAPHS, MOVTPH, MOVTPHS**

Example 4.14.36.1

```
MOVSPH A0, MR, *R3+R5
```

Load the content of byte address created by adding **R3** and **R5** to **MR** register. At the same time, subtract **PH** register from accumulator **A0**, store result in **A0**.

4.14.35 MOVSPHS Move String With Subtract From PH

Syntax

[label]	***name***	*dest, src, src1*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	MOVSPHS	**A***n***, MR,** {*adrs*}	Table 4–46		Table 4–46	1b

Execution

A*n* ⇐ **A***n* (second word) – **PH**
MR ⇐ contents of {*adrs}*
PC ⇐ **PC** + *w*

Flags Affected

TAG, OF, **SF**, **ZF**, **CF** are set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOVSPHS A*n***, MR,** {*adrs*}	0	1	1	0	0	1	1	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															

Description

Move data memory word string to **MR**, subtract **PH** from second word **A***n* string. Store result in **A***n*. Certain restrictions apply to the use of this instruction when interrupts are occuring on the background. See Section 4.8 for more details.

See Also

MOVSPH, MOVAPH, MOVAPHS, MOVTPH, MOVTPHS

Example 4.14.37.1

```
MOVSPHS A0, MR, *R3+R5
```

Load the content of byte address created by adding **R3** and **R5** to **MR** register. At the same time, subtract **PH** register from second word of **A0** string, store result in **A0** string.

4.14.36 MOVT Move Tag From Source to Destination

Syntax

[label]	***name***	*dest, src*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	MOVT	{*adrs*}, **TF***n*	Table 4–46		Table 4–46	5

Execution

dest ⇐ *src*
PC ⇐ **PC** + *w*

Flags Affected **None**

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOVT {*adrs*}**, TF***n*	1	1	0	1	0	1	1	1	*fig*	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															

Description Move **TF***n* from **STAT** register to memory tag. All addressing modes are available.

See Also **MOVU, MOV, MOVT, MOVB, MOVBS, MOVS**

Example 4.14.32.1

```
MOVT *R3++, TF2
```

Copy the **TF2** flag bit to the 17th bit of the word pointed by **R3**. Increment **R3** by 2.

4.14.37 MOVU Move Data Unsigned

Syntax

[label]	***name***	*dest, src* [, *mod*]	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	MOVU	**MR, A***n*[**~**] [**,** *next A*]	1	1	n_R+3	3
	MOVU	**MR,** {*adrs*}	Table 4–46		Table 4–46	5

Execution
[premodify **AP** if *mod* specified]
dest ⇐ *src*
PC ⇐ **PC** + *w*

Flags Affected
src is {*adrs*} **TAG** bit is set accordingly
UM is set to 1

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOVU MR, A*n*[**~**] [**,** *next A*]	1	1	1	0	0	*next A*		***A****n*		1	0	1	1	1	0	**A~**	0
MOVU MR, {*adrs*}	1	1	0	1	1	1	0	0	1	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															

Description
Copy value of *src* to *dest*. Premodification of accumulator pointers is allowed with some operand types.

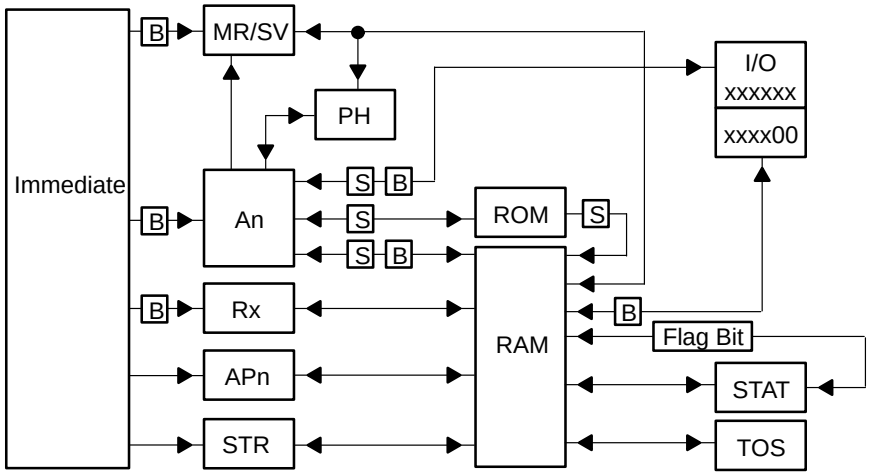
Syntax	**Description**
MOVU MR, A*n*[**~**] [**,** *next A*]	Move **A***n*[**~**] to **MR** register in unsigned multiplier mode
MOVU MR, {*adrs*}	Move data memory word to **MR**, reset multiplier signed mode

See Also
MOV, MOVB, MOVT, MOVBS, MOVS

Example 4.14.33.1
```
MOVU MR, A0~, ++A
```
Preincrement accumulator pointer **AP0**. Copy the content of accumulator **A0~** to **MR** register.

Example 4.14.33.2
```
MOVU MR, *R3
```
Copy the value pointed by R3 to MR.

Figure 4–8. Valid Moves/Transfer in MSP50P614/MSP50C614 Instruction Set



NOTE: B = Byte move possible.
S = String move possible.
R5 can be moved to Rx, *An[–]* to *An[–]*

4.14.38 MUL Multiply (Rounded)

Syntax

[label]	***name***	*src* [, *mod*]	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	MUL	**A***n*[**~**] [**,** *next A*]	1	1	n_R+3	3
	MUL	{*adrs*}	Table 4–46		Table 4–46	5

Execution

[premodify **AP** if *mod* specified]
PH,**PL** ⇐ **MR** * *src*
PC ⇐ **PC** + *w*

Flags Affected

src is **A***n* : **OF**, **SF**, **ZF**, **CF** are set accordingly
src is {*adrs*}: **TAG** bit is set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MUL A*n*[**~**] [**,** *next A*]	1	1	1	0	0	*next A*		***A****n*		1	1	1	1	0	0	**A~**	0
MUL {*adrs*}	1	1	0	1	1	1	0	1	1	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															

Description

Multiply **MR** and *src*. The 16 MSBs of the 32–bit product are stored in the the **PH** register. The contents of the accumulator are not changed. The upper 16 bits of the result are rounded for **MUL A**n, but not for **MUL** {*adrs*}. Pre–modify the accumulator pointer if specified.

Syntax	**Description**
MUL A*n*[**~**] [*, next A*]	Multiply **MR** by **A***n*[**~**] word, store result in **A***n*[**~**]†
MUL {*adrs*}	Multiply **MR** by data memory word‡

† Round upper 16 bits
‡ No status change

See Also **MULR, MULAPL, MULSPL, MULSPLS, MULTPL, MULTPLS, MULAPL**

Example 4.14.38.1 `MUL A0~, --A`

Predecrement accumulator pointer **AP0**. Multiply **MR** with accumulator **A0~** and store upper 16 bits of the result (rounded) **PH**. Accumulator A0~ is left unchanged.

Example 4.14.38.2 `MUL *R3--`

Multiply **MR** with the value pointed at by **R3** and store the upper 16 bits of the result (rounded) into **PH**. Decrement **R3** by 2.

4.14.39 MULR Multiply (Rounded) With No Data Transfer

Syntax

[label]	***name***	*src*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	MULR	{adrs}	Table 4–0–46		Table 4–0–46	5

Execution **PH**,**PL** ⇐ **MR** * *src*
PC ⇐ **PC** + 1

Flags Affected **TAG** bit is set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MULR [adrs]	1	1	0	1	1	1	0	1	0	adrs							
	x	dma16 (for direct) or offset16 (long relative) (see Section 4.13)															

Description Perform multiplication of multiply register (**MR**) and effective data memory value, add 08x00 to the product. The 16 MSBs of the 32-bit product are stored in the product high (**PH**) register. No status change. Round upper 16 bits.

See Also **MULS, MUL, MULAPL, MULSPL, MULSPLS, MULTPL, MULTPLS, MULAPL**

Example 4.14.39.1 `MULR *R0++`
Multiply **MR** with the content of data memory location pointed by **R0** and store the rounded upper 16 bits of the result in **PH**. Increment **R0** by 2.

4.14.40 MULS Multiply String With No Data Transfer

Syntax

[label]	***name***	*src*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	MULS	A*n*[~]	n_S+3	1	n_R+3	3

Execution **PH**,**PL** ⇐ **MR** * *src* string
PC ⇐ **PC** + 1

Flags Affected None

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MULS A*n*[~]	1	1	1	0	0	1	1	**A***n*		1	1	1	1	0	0	**A~**	0

Description Multiply **MR** and the value in *src*. The 16 MSBs of the ((n_S+3) x 16) –bit product are stored in the **PH** register. The value in *src* is unchanged and the value in **PL** is ignored. This instruction rounds the upper 16 bits. Note that **A***n* is a string of length n_S+2, where n_S is the value in **STR** register.

See Also **MUL, MULR, MULAPL, MULSPL, MULSPLS, MULTPL, MULTPLS, MULAPL**

Example 4.14.40.1
```
MULS A0
```
Multiply **MR** with **A0** and store the upper 16 bits (with rounding) to **PH** register.

4.14.41 MULAPL Multiply and Accumulate Result

Syntax

[label]	***name***	*dest, src* [*, mod*]	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	MULAPL	**A***n***,** {*adrs*}	Table 4–46		Table 4–46	1b
	MULAPL	**A***n*[**~**]**, A***n*[**~**] [**,** *next A*]	1	1	n_R+3	3

Execution

[premodify **AP** if *mod* specified]
PH,**PL** ⇐ **MR** * *src*
dest ⇐ *dest* + **PL**
PC ⇐ **PC** + 1

Flags Affected

OF, **SF**, **ZF**, **CF** are set accordingly
src is {*adrs*}: **TAG** bit is set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MULAPL A*n***,** {*adrs*}	0	1	1	0	1	1	0	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MULAPL A*n*[**~**]**, A***n*[**~**], [*next A*]	1	1	1	0	0	*next A*		**A***n*		1	1	0	0	1	0	**A~**	**~A**

Description

Perform multiplication of multiply register (**MR**) and value of *src*. The 16 MSBs of the 32-bit product are stored in the product high (**PH**) register. The 16 LSBs of the product (contained in product low (**PL**) register) added to *dest*. Certain restriction applies to the use of this instruction when interrupts are occuring in the background. See Section 4.8 for more detail.

Syntax	**Description**
MULAPL {*adrs*}	Multiply **MR** by RAM word, add **PL** to **A***n*
MULAPL A*n*[**~**]**, A***n*[**~**] [**,** *next A*]	Multiply **MR** by An[~] word, add **PL** to **A***n*[**~**]

See Also **MULAPLS, MULSPL, MULSPLS, MULTPL, MULTPLS**

Example 4.14.41.1 `MULAPL A0, *R3++`

Multiply **MR** with the content of data memory word stored at byte location pointed by **R3**, add **PL** to accumulator **A0**, and store result in accumulator **A0**. Increment **R3** by 2.

Example 4.14.41.2 `MULAPL A2, A2~, --A`

Multiply **MR** register to accumulator **A2~**, add **PL** to accumulator **A2**, and store result to accumulator **A2**.

4.14.42 MULAPLS Multiply String and Accumulate Result

Syntax

[label]	***name***	*dest, src* [*, mod*]	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	MULAPLS	**A***n***,** {*adrs*}	Table 4–46		Table 4–46	1b
	MULAPLS	**A***n*[~]**, A***n*[~]	n_S+3	1	n_R+3	3

Execution

PH,**PL** ⇐ **MR** * *src*
dest ⇐ *dest* + **PL**
PC ⇐ **PC** + 1

Flags Affected

OF, **SF**, **ZF**, **CF** are set accordingly
src is {*adrs*} : **TAG** bit is set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MULAPLS A*n***,** {*adrs*}	0	1	1	0	1	1	1	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MULAPL S A*n*[~]**, A***n*[~], [*next A*]	1	1	1	0	0	1	1	**A***n*		1	1	0	0	1	0	**A~**	**~A**

Description

Perform multiplication of multiply register (**MR**) and value of *src*. The 16 MSBs of the $((n_S + 3) \times 16)$-bit product are stored in the product high (**PH**) register. The 16 LSBs of the product (contained in product low (**PL**) register) added to *dest* string.

Syntax	**Description**
MULAPLS {*adrs*}	Multiply **MR** by RAM string, add **PL** to **A***n*
MULAPLS A*n*[~]**, A***n*[~] [**,** *next A*]	Multiply **MR** by An[~] string, add **PL** to **A***n*[~]

See Also **MULAPL, MULSPL, MULSPLS, MULTPL, MULTPLS**

Example 4.14.42.1 `MULAPLS A0, *R3++`

Multiply **MR** with the content of data memory word string store at byte location pointed by **R3,** add accumulator string **A0** to **PL**, and store result in accumulator **A0** string. Increment **R3** by 2.

Example 4.14.42.2 `MULAPLS A2, A2~, --A`

Multiply **MR** register to accumulator **A2~**, add accumulator string **A2** to **PL** and store result to accumulator **A2**.

4.14.43 MULSPL Multiply and Subtract PL From Accumulator

Syntax

[label]	***name***	*dest, src* [*, mod*]	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	MULSPL	**A***n***,** {*adrs*}	Table 4–46		Table 4–46	1b
	MULSPL	**A***n*[**~**]**, A***n*[**~**] [, *next A*]	1	1	n_R+3	3

Execution

[premodify **AP** if *mod* specified]
PH,**PL** ⇐ **MR** * *src*
dest ⇐ *dest* – **PL**
PC ⇐ **PC** + 1

Flags Affected

OF, **SF**, **ZF**, **CF** are set accordingly
src is {*adrs*}: **TAG** bit is set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MULSPL A*n***,** {*adrs*}	0	1	1	1	1	1	1	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MULSPL A*n*[**~**]**, A***n*[**~**], [*next A*]	1	1	1	0	0	*next A*		**A***n*		1	1	0	0	0	0	**A~**	**~A**

Description

Perform multiplication of multiply register (**MR**) and value of *src*. The 16 MSBs of the 32-bit product are stored in the product high (**PH**) register. The 16 LSBs of the product (contained in product low (**PL**) register) are subtracted from *dest*. Certain restrictions apply to the use of this instruction when interrupts are occuring in the background. See Section 4.8 for more details.

Syntax	**Description**
MULSPL {*adrs*}	Multiply **MR** by RAM word, substract **PL** to **A***n*
MULSPL A*n*[**~**]**, A***n*[**~**] [**,** *next A*]	Multiply **MR** by An[~] word, substract **PL** to **A***n*[**~**]

See Also **MULSPLS, MULTPL, MULTPLS, MULAPL, MULAPLS**

Example 4.14.43.1 `MULSPL A0, *R3++`

Multiply **MR** with the contents of **R3**, subtract **PL** from accumulator **A0**. and store result in accumulator **A0** post-increment. Post-increment **R3** by 2.

Example 4.14.43.2 `MULSPL A2, A2~, --A`

Predecrement accumulator pointer **AP2**. Multiply **MR** register to accumulator **A2~**, subtract **PL** from accumulator **A2**, and store result to accumulator **A2**.

4.14.44 MULSPLS Multiply String and Subtract PL From Accumulator

Syntax

[label]	***name***	*dest, src*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	MULSPLS	**A***n***,** {*adrs*}	Table 4–46		Table 4–46	1b
	MULSPLS	**A***n*[~]**, A***n*[~]	n_S+3	1	n_R+3	3

Execution

PH,**PL** ⇐ **MR** * *src*
dest ⇐ *dest* – **PL**
PC ⇐ **PC** + 1

Flags Affected

OF, **SF**, **ZF**, **CF** are set accordingly
src is {*adrs*}: **TAG** bit is set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MULSPLS A*n***,** {*adrs*}	0	1	1	1	1	1	1	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MULSPL S A*n*[~]**, A***n*[~]	1	1	1	0	0	1	1	**A***n*		1	1	0	0	0	0	**A~**	**~A**

Description

Perform multiplication of multiply register (**MR**) and value of *src*. The 16 MSBs of the $((n_S + 3) \times 16)$-bit product are stored in the product high (**PH**) register. The 16 LSBs of the product (contained in product low (**PL**) register) subtracted from *dest* string.

Syntax	**Description**
MULSPLS {*adrs*}	Multiply **MR** by data memory string, subtract **PL** from **A***n*
MULSPLS A*n*[~]**, A***n*[~]	Multiply **MR** by **A***n*[~] string, subtract **PL** from **A***n*[~]

See Also **MULSPL, MULTPL, MULTPLS, MULAPL, MULAPLS**

Example 4.14.44.1 `MULSPLS A0, *R3++`

Multiply **MR** with the contents of **R3**, subtract **PL** from accumulator string **A0**, and store result in accumulator string **A0**. Increment **R3** by 2.

Example 4.14.44.2 `MULSPLS A2, A2~`

Multiply **MR** register to accumulator string **A2~**, subtract **PL** from accumulator string **A2**, and store result to accumulator string **A2**.

4.14.45 MULTPL Multiply and Transfer PL to Accumulator

Syntax

[label]	***name***	*dest, src* [*, mod*]	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	MULTPL	**A***n***,** {*adrs*}	Table 4–46		Table 4–46	1b
	MULTPL	**A***n*[**~**]**, A***n*[**~**] [, *next A*]	1	1	n_R+3	3

Execution

[premodify **AP** if *mod* specified]
PH,**PL** ⇐ **MR** * *src*
A*n* ⇐ **PL**
PC ⇐ **PC** + 1

Flags Affected

OF, **SF**, **ZF**, **CF** are set accordingly
src is {*adrs*}: **TAG** bit is set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MULTPL A*n***,** {*adrs*}	0	1	1	0	0	0	0	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MULTPL A*n*[**~**]**, A***n*[**~**], [*next A*]	1	1	1	0	0	*next A*		**A***n*		1	1	0	1	1	0	**A~**	**~A**

Description

Perform multiplication of multiply register (**MR**) and value of *src*. The 16 MSBs of the 32-bit product are stored in the product high (**PH**) register. The 16 LSBs of the product (contained in product low (**PL**) register) are stored in **A***n*. Certain restrictions apply to the use of this instruction when interrupts are occuring in the background. See Section 4.8 for more detail.

Syntax	**Description**
MULTPL {*adrs*}	Multiply **MR** by data memory word, move **PL** to **A***n*
MULTPL A*n*[**~**]**, A***n*[**~**] [**,** *next A*]	Multiply **MR** by An[~] word, move **PL** to **A***n*[**~**]

See Also **MULTPLS, MULAPL, MULAPLS, MULSPL, MULSPLS**

Example 4.14.45.1 `MULTPL A0, *R3++`

Multiply the contents of **R3** with **MR** register and store **PL** in accumulator **A0**. Increment **R3** by 2.

Example 4.14.45.2 `MULTPL A2, A2~, --A`

Multiply **MR** register to accumulator **A2~** and store **PL** to accumulator **A2**.

4.14.46 MULTPLS Multiply String and Transfer PL to Acumulator

Syntax

[label]	***name***	*dest, src*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	MULTPLS	**A***n***,** {*adrs*}	Table 4–46		Table 4–46	1b
	MULTPLS	**A***n*[**~**]**, A***n*[**~**]	n_S+3	1	n_R+3	3

Execution

PH, **PL** ⇐ **MR** * *src*
A*n* ⇐ **PL**
PC ⇐ **PC** + 1

Flags Affected

OF, **SF**, **ZF**, **CF** are set accordingly
src is {*adrs*}: **TAG** bit is set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MULSPLS A*n***,** {*adrs*}	0	1	1	0	0	0	1	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MULSPL S A*n*[~]**, A***n*[~]	1	1	1	0	0	1	1	**A***n*		1	1	0	1	1	0	**A~**	**~A**

Description

Perform multiplication of multiply register (**MR**) and value of *src* string. The 16 MSBs of the $((n_S + 3) \times 16)$-bit product are stored in the product high (**PH**) register. The 16 LSBs of the product (contained in product low (**PL**) register) stored in **A***n* string.

Syntax	**Description**
MULTPLS A*n***,** {*adrs*}	Multiply **MR** by effective data memory string, move **PL** to **A***n*
MULTPLS A*n*[**~**]**, A***n*[**~**]	Multiply **MR** by **A***n*[**~**] string, move **PL** to **A***n*[**~**]

See Also **MULTPL, MULAPL, MULAPLS, MULSPL, MULSPLS**

Example 4.14.46.1 `MULTPLS A0, *R3++`

Multiply the contents of **R3** with **MR** register and store **PL** in accumulator string **A0**. Increment **R3** by 2.

Example 4.14.46.2 `MULTPLS A2, A2~`

Multiply **MR** register to accumulator string **A2~** and store **PL** to accumulator string **A2**.

4.14.47 NEGAC Two's Complement Negation of Accumulator

Syntax

[label]	***name***	*dest, src [,mod]*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	NEGAC	**A***n*[**~**]**,** **A***n*[**~**] [, *next A*]	n_S+3	1	n_R+3	3

Execution [premodify **AP** if *mod* specified]
dest ⇐ *–src*
PC ⇐ **PC** + 1

Flags Affected **OF**, **SF**, **ZF**, **CF** are set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
NEGAC A*n*[**~**]**, A***n*[**~**] [**,** *next A*]	1	1	1	0	0	*next A*		**A***n*		0	0	0	0	0	0	**A~**	**~A**

Description Perform two's complement negation of *src* accumulator and store result in *dest* accumulator.

See Also **NEGACS, SUB, SUBB, SUBS, ADD, ADDB, ADDS, NOTAC, NOTACS**

Example 4.14.47.1
```
NEGAC A3~, A3, --A
```
Predecrement accumulator pointer **AP3**. Negate accumulator **A3** and store result in accumulator **A3~**.

4.14.48 NEGACS Two's Complement Negation of Accumulator String

Syntax

[label]	***name***	*dest, src*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	NEGACS	**A***n*[**~**]**, A***n*[**~**]	n_S+3	1	n_R+3	3

Execution

dest ⇐ *–src*
PC ⇐ **PC** + 1

Flags Affected

OF, **SF**, **ZF**, **CF** are set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MULSPL S A*n*[**~**]**, A***n*[**~**]	1	1	1	0	0	1	1	**A***n*		0	0	0	0	0	0	**A~**	**~A**

Description

Perform two's complement negation of *src* accumulator string and store result in *dest* accumulator string.

See Also

NEGAC, SUB, SUBB, SUBS, ADD, ADDB, ADDS, NOTAC, NOTACS

Example 4.14.48.1

```
NEGACS A3~, A3
```

Negate accumulator string **A3** and store result in accumulator string **A3~**.

4.14.49 NOP No Operation

Syntax

[label]	***name***	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	NOP	1	1	n_R+3	9d

Execution **PC** ⇐ **PC** + 1 (No operation)

Flags Affected **None**

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
NOP	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Description This instruction performs no operation. It consumes 1 clock of execution time and 1 word of program memory.

See Also **RPT**

Example 4.14.49.1 `NOP`
Consumes 1 clock cycle.

4.14.50 NOTAC One's Complement Negation of Accumulator

Syntax

[label]	***name***	*dest, src* [*, mod*]	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	NOTAC	**A***n*[**~**]**, A***n*[**~**] [, *next A*]	1	1	n_R+3	3

Execution

[premodify **AP** if *mod* specified]
dest ⇐ NOT *src*
PC ⇐ **PC** + 1

Flags Affected

OF, **SF**, **ZF**, **CF** are set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
NOTAC A*n*[~]**, A***n*[~] [**,** *next A*]	1	1	1	0	0	*next A*		**A***n*		0	0	0	0	1	0	**A~**	**~A**

Description

Premodify accumulator pointer if specified. Perform one's complement of *src* accumulator and store result in *dest* accumulator.

See Also

NOTACS, AND, ANDB, ANDS, OR, ORB, ORS, XOR, XORB, XORS, NEGAC, NEGACS

Example 4.14.50.1

```
NOTAC A3~, A3, --A
```

Predecrement accumulator pointer **AP3**. One's complement (invert bits) accumulator **A3** and put result in accumulator **A3~**.

4.14.51 NOTACS One's Complement Negation of Accumulator String

Syntax

[label]	***name***	*dest, src*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	NOTACS	**A***n*[**~**], **A***n*[**~**]	n_S+2	1	n_R+2	3

Execution *dest* ⇐ NOT *src*
PC ⇐ **PC** + 1

Flags Affected **OF**, **SF**, **ZF**, **CF** are set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
NOTACS A*n*[**~**], **A***n*[**~**]	1	1	1	0	0	1	1	**A***n*		0	0	0	0	1	0	**A~**	**~A**

Description Perform one's complement of *src* accumulator string and store result in *dest* accumulator string.

See Also **NOTAC, AND, ANDB, ANDS, OR, ORB, ORS, XOR, XORB, XORS, NEGAC, NEGACS**

Example 4.14.51.1 `NOTACS A3~, A3`
Take the one's complement (invert bits) of the accumulator string **A3** and put result in accumulator string **A3~**.

4.14.52 OR Bitwise Logical OR

Syntax

[label]	***name***	*dest, src [, src1] [, mod]*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	OR	**A***n***,** {*adrs*}	Table 4–46		Table 4–46	1b
	OR	**A***n*[**~**]**,** **A***n*[**~**]**,** *imm16* [, *next A*]	2	2	N/R	2b
	OR	**A***n*[**~**]**,** **A***n***~,** **A***n* [**,** *next A*]	1	1	n_R+3	3
	OR	**TF***n***,** {*flagadrs*}	1	1	N/R	8a
	OR	**TF***n***,** {*cc*} [**,** **R***x*]	1	1	n_R+3	8b

Execution

[premodify **AP** if *mod* specified]
dest ⇐ *dest* OR *src1* (for two operands)
dest ⇐ *src* OR *src1* (for three operands)
PC ⇐ **PC** + *w*

Flags Affected

dest is **A***n*: **OF**, **SF**, **ZF**, **CF** are set accordingly
dest is **TF***n*: **TF***n* bits in **STAT** register are set accordingly
src is {*adrs*}: **TAG** bit is set accordingly
src is {*flagadrs*}: **TAG** bit is set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OR A*n*, {*adrs*}	0	1	0	0	0	0	0	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
OR A*n*[**~**]**,** **A***n*[**~**]**,** *imm16* [, *next A*]	1	1	1	0	0	*next A*		**A***n*		1	0	0	0	0	1	**A~**	**~A**
OR A*n*[**~**]**,** **A***n***~,** **A***n* [, *next A*]	1	1	1	0	0	*next A*		**A***n*		0	1	0	0	1	0	**A~**	**~A**
OR TF*n*, {*flagadrs*}	1	0	0	1	1	*fig*	Not	0	1	0	*flagadrs*						
OR TF*n*, {*cc*} [**,** **R***x*]	1	0	0	1	0	*fig*	Not	*cc*					**R***x*			0	1

Description

Bitwise OR of *src* and *dest*. Result is stored in *dest*. If three operands are specified then logical OR *src* and *src1*, store result in *dest*. Premodification of accumulator pointers are allowed with some operand types.

Syntax	**Description**
OR A*n***,** {*adrs*}	OR RAM word to **A***n*
OR A*n*[**~**]**,** **A***n*[**~**]**,** *imm16* [, *next A*]	OR immediate word to **A***n*[**~**], store result in **A***n*[**~**]
OR A*n*[**~**]**,** **A***n***~,** **A***n* [, *next A*]	OR **A***n* word to **A***n***~** word, store result in **A***n*[**~**]
OR TF*n***,** {*flagadrs*}	OR **TF***n* with memory tag, store result in **TF***n* bit in **STAT**
OR TF*n*, {*cc*} [**,** **R***x*]	**OR** test condition with **TF***n* bit in **STAT** register· **R***x* must be provided if *cc* is one of {**RZP**, **RNZP**, **RLZP**, **RNLZP**} to check if the selected **R***x* is zero or negative. **R***x* should not be provided for other conditionals.

See Also **ORB, ORS, AND, ANDS, XOR, XORS, NOTAC, NOTACS**

Example 4.14.52.1 `OR A0, *R0++R5`
OR accumulator **A0** with the value in data memory address stored in **R0** and store result in accumulator **A0**, Add **R5** to **R0** after execution.

Example 4.14.52.2 `OR A1, A1, 0xF0FF, ++A`
Preincrement pointer **AP1**. OR immediate 0xF0FF to accumulator **A1**. Store result in accumulator **A1**.

Example 4.14.52.3 `OR A1, A1~, A1, --A`
Pre–decrement accumulator pointer **AP1**. OR accumulator **A1** to accumulator **A1~**, put result in **A1**.

Example 4.14.52.4 `OR TF1, *R6+0x22`
OR **TF1** bit in **STAT** with tag bit (17^{th} bit) at relative flag address 0x22 relative to **R6** (i.e., **R6**+0x22), store result in **TF1** flag in **STAT**.

Example 4.14.52.5 `OR TF1, ZF`
OR **ZF** flag in **STAT** register with to **TF1**, put result in **TF1** bit in **STAT**.

Example 4.14.52.6 `OR TF2, RZP, R2`
OR **TF2** with the condition code **RZP** (**R***x*=0 flag) for **R2**, and store result in **TF2**. If the content of **R2** is zero then **RZP** condition becomes true, otherwise false. **TF2** bit in **STAT** is modified based on this result.

4.14.53 ORB Bitwise OR Byte

Syntax

[label]	***name***	*dest, src*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	ORB	**A***n***,** *imm8*	1	1	N/R	2a

Execution *dest* ⇐ *dest* OR *src*
PC ⇐ **PC** + 1

Flags Affected **OF**, **SF**, **ZF**, **CF** are set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ORB A*n***,** *imm8*	1	0	1	0	1	0	0	**A***n*		*imm8*							

Description Bitwise OR byte of *src* and *dest*. Result is stored in *dest*. Only lower 8 bits of accumulator is affected.

See Also **OR, ORS, AND, ANDS, XOR, XORS, NOTAC, NOTACS**

Example 4.14.53.1
```
ORB A2, 0x45
```
OR 0x45 immediate to accumulator **A2** lower 8 bits.

4.14.54 ORS Bitwise OR String

Syntax

[label]	***name***	*dest, src* [, *src1*]	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	ORS	**A***n***,** {*adrs*}	Table 4–46		Table 4–46	1b
	ORS	**A***n*[~]**, A***n*[~]**,** *pma16*	n_S+4	2	N/R	2b
	ORS	**A***n*[~]**, A***n*~**, A***n*	n_S+2	1	n_R+2	3

Execution

dest ⇐ *dest* OR *src* (for two operands)
dest ⇐ *src1* OR *src* (for three operands)
C ⇐ **PC** + *w*

Flags Affected

dest is **A***n*: **OF**, **SF**, **ZF**, **CF** are set accordingly
src is {*adrs*}: **TAG** bit is set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ORS A*n***,** {*adrs*}	0	1	0	0	0	0	1	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
ORS A*n*[~]**, A***n*[~]**,** *pma16*	1	1	1	0	0	1	1	**A***n*		1	0	0	0	0	1	**A~**	**~A**
ORS A*n*[~]**, A***n*~**, A***n*	1	1	1	0	0	1	1	**A***n*		0	1	0	0	1	0	**A~**	**~A**

Description

Bitwise OR of *src* and *dest*. Result is stored in *dest*. If three operands are specified then logical OR *src1* and *src*, store result in *dest*.

Syntax	**Description**
ORS A*n***,** {*adrs*}	OR RAM string to **A***n* string
ORS A*n*[~]**, A***n*[~]**,** *pma16*	OR ROM string to **A***n*[~] string, store result in **A***n*[~] string
ORS A*n*[~]**, A***n*~**, A***n*	OR **A***n* string to **A***n*~ string, store result in **A***n*[~] string

See Also **OR, ORB, AND, ANDS, XOR, XORS, NOTAC, NOTACS**

Example 4.14.54.1

```
ORS A0, *R2
```

OR data memory string beginning at address in **R2** to accumulator string **A0**. Result stored in accumulator string **A0**.

Example 4.14.54.2

```
ORS A0, A0~, 0x13F0
```

OR program memory string beginning at address in 0x13F0 to accumulator string **A0~**, put result in accumulator string **A0**. Note that the address 0x13F2 is a program memory address.

Example 4.14.54.3

```
ORS A0, A0~, A0
```

OR accumulator string **A0** to accumulator string **A0~**, put result in accumulator string **A0**.

4.14.55 OUT Output to Port

Syntax

[label]	***name***	*dest, src*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	OUT	*port4*, {*adrs*}	Table 4–46		n_R+3	6a
	OUT	*port6*, **A***n*[**~**]	Table 4–46		n_R+3	6a

Execution *port4* or *port6* ⇐ *src*
PC ⇐ **PC** + *w*

Flags Affected **XSF**, **XZF** are set accordingly
src is {*adrs*}: **TAG** bit is set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OUT *port4*, {*adrs*}	1	1	0	0	1	port4				*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
OUT *port6*, **A***n*[**~**]	1	1	1	0	1	1	0	**A***n*		*port6*						1	**~A**

Description Output to I/O port. Words (16 bits) in memory can be output to one of 16 port addresses. Words (16 bits) in the accumulators can be output to these same 16 port addresses or to an additional 48 port addresses. Note that, *port4* address is multipled by 4 to get the actual port address.

See Also **OUTS, IN, INS**

Example 4.14.55.1
```
OUT 3, * 0x0200 * 2
```
Outputs the content of word memory location value stored in 0x0200 to I/O port at location 0x0C (**PBDIR** port). Note that, address 3 converts to 3 * 4 = 0xc.

4.14.56 OUTS Output String to Port

Syntax

[label]	***name***	*dest, src*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	OUTS	*port6*, **A***n*[**~**]	n_R+2	1	n_R+2	6b

Execution *port6* ⇐ *src*
PC ⇐ PC + 1

Flags Affected **XSF**, **XZF** are set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OUTS *port6*, **A***n*[**~**]	1	1	1	0	1	1	1	**A***n*		*port6*						1	**~A**

Description Output to I/O port. Word in the accumulator string can be output to one of 64 port addresses. String operation writes several consecutive ports starting from *port6* specified in the instruction.

See Also **OUT, IN, INS**

Example 4.14.56.1
```
OUTS 0x04, A3
```
Put the content of acccumulator string **A3** to I/O port string address 0x04 (**PADIR** port). Note that, based on string length, other consecutive ports may also be written.

4.14.57 RET Return From Subroutine (CALL, C*cc*)

Syntax

[label]	***name***	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	RET	1	1	N/R	5

Execution

PC ⇐ **TOS**
TOS ⇐ ***R7**
R7 ⇐ **R7** – 2

Flags Affected None

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RET	1	1	0	1	1	1	1	0	0	0	1	1	1	1	1	1	0

Description Return from call or vectored call. Pop stack to program counter, continue execution. Returns from subroutine calls (**CALL**, **C*cc*** instructions) and interrupts are different because of the way each process is handled. In order to prevent execution pipeline problems the interrupt return (**IRET**) instruction uses two cycles and the Return (**RET**) instruction cannot immediately follow a **CALL**, i.e., **RET** followed by a **RET** should not be allowed.

See Also **CALL, C*cc*, IRET**

Example 4.14.57.1 RET
Returns from subroutine. A **CALL** or **C*cc*** instruction must have executed before.

4.14.58 RFLAG Reset Memory Flag

Syntax

[label]	***name***	*src*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	RFLAG	{*flagadrs*}	1	1	N/R	8a

Execution memory flag bit at {*flagadrs*} data memory location ⇐ 0
PC ⇐ **PC** + 1

Flags Affected None

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RFLAG *{flagadrs}*	1	0	0	1	0	0	0	0	1	1	*flagadrs*						

Description Reset flag at addressed memory location to 0.{*flagadrs*} includes two groups of memory flag addresses: global flags, which are the first 64 word locations in RAM; and relative flags, which are 64 locations relative to the page register (**R6**). Flag address {*flagadrs*} only addresses the 17th bit. (See section 4.3.7 for more information)

See Also **SFLAG**, **STAG**, **RTAG**

Example 4.14.58.1 `RFLAG *0x21`
Resets the flag bit at RAM byte location 0x0042 to zero.

Example 4.14.58.2 `RFLAG *R6 + 0x0002`
Resets the flag bit at RAM byte location 0x0084 to zero. Assume **R6** = 0x0080. The **R6** register is represented in bytes, but the 0x0002 is represented in words. Thus, 0x0080 bytes plus 0x0002 words (or 0x0004 bytes) equals 0x0084 (bytes).

4.14.59 RFM Reset Fractional Mode

Syntax

[label]	***name***	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	RFM	1	1	N/R	9d

Execution **STAT.FM** ⇐ 0
PC ⇐ **PC** + 1

Flags Affected None

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RFM	1	1	1	1	1	1	1	1	0	1	1	0	1	0	0	0	0

Description Resets fractional mode. Clears bit 3 in status register (**STAT**). Disable multiplier shift mode for unsigned fractional or integer arithmetic.

See Also **SFM**

Example 4.14.59.1 `RFM`
Resets the fractional mode. Clears **FM** bit of **STAT**.

4.14.60 ROVM Reset Overflow Mode

Syntax

[label]	***name***	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	ROVM	1	1	N/R	9d

Execution

STAT.OM ⇐ 0
PC ⇐ **PC** + 1

Flags Affected None

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RFM	1	1	1	1	1	1	1	1	0	1	1	0	1	0	0	0	0

Description Resets overflow mode in status register bit 2 (the **OM** bit). Disable ALU saturation output (normal mode).

See Also **SOVM**

Example 4.14.60.1 `ROVM`
Resets the overflow mode to zero.

4.14.61 RPT Repeat Next Instruction

Syntax

[label]	***name***	*src*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	RPT	$\{adrs\}_8$	Table 4–46		N/R	5
	RPT	*imm8*	1	1	N/R	9b

Execution

IF RPT $\{adrs\}_8$
load *src* to repeat counter.
ELSE
load *imm8* to repeat counter.
(*mask interrupt*)
repeat next instruction (repeat counter value + 2) times.
PC ⇐ **PC** + *w* (next instruction)+1

Flags Affected None

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RPT $\{adrs\}_8$	1	1	0	1	1	1	1	1	0	*adrs*							
RPT *imm8*	1	1	1	1	1	1	0	0	0	*imm8*							

Description Loads *src* value to repeat counter. Execute next instruction *src* value + 2 times. Interrupts are queued during **RPT** instruction. Queued interrupts are serviced after execution completes.

Syntax	**Description**
RPT $\{adrs\}_8$	Load data memory byte to repeat counter, repeat next instruction
RPT *imm8*	Load immediate byte to repeat counter, repeat next instruction

See Also **BEGLOOP, ENDLOOP**

Example 4.14.61.1

```
RPT *0x0100 * 2
MOV *R1++, A0, ++A
```

Loads the repeat counter with value stored in word data memory location 0x0100. Only 8 bits of data from this location are used. The next instruction stores content of **A0** to data memory address pointed by **R1**. Since **R1** post increments and **A0** preincrements in this instruction, the overall effect of executing this instruction with **RPT** is to store accumulator contents to consecutive data memory locations. See **MOV** instruction for detail of various syntax of **MOV** instruction.

Example 4.14.61.2

```
RPT 200
NOP
```

Repeat the **NOP** instruction 202 times (provided the next instruction is repeatable). This causes 203 instruction cycle delay (including 1 cycle for the **RPT** instruction).

4.14.62 RTAG Reset Tag

Syntax

[label]	***name***	*dest*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	RTAG	{*adrs*}	Table 4–46		Table 4–46	5

Execution memory tag bit at {*adrs*} data memory location ⇐ 0
PC ⇐ **PC** + 1

Flags Affected None

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RTAG {*adrs*}	1	1	0	1	0	1	1	0	1	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															

Description Resets tag bit at addressed memory location. All addressing modes are available. Note that this instruction accesses only the 17th bit of the RAM location. For odd RAM byte addresses, the least significant bit is ignored.

See Also **STAG**, **RFLAG**, **SFLAG**

Example 4.14.62.1 `RTAG * 0x0200 * 2`
Reset the tag bit of data memory word location to 0. Note that this operation can also be done with **RFLAG** by loading the R6 register with * 0200 * 2.

Example 4.14.62.2 `RTAG *R6+0x0002`
Reset the tag bit of RAM location 0x0082. Assume **R6** = 0x0080. Unlike the **SFLAG** and **RFLAG** instructions, the argument of the **STAG**/**RTAG** instruction is interpreted as bytes.

Example 4.14.62.3 `RTAG *R6+0x0003`
Reset the tag bit of RAM location 0x0082. Assume **R6** = 0x0080.

4.14.63 RXM Reset Extended Sign Mode

Syntax

[label]	***name***	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	RXM	1	1	N/R	9d

Execution

STAT.XM ⇐ 0
PC ⇐ **PC** + 1

Flags Affected None

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RXM	1	1	1	1	1	1	1	1	0	1	0	1	1	0	0	0	0

Description Reset extended sign mode status register bit 0 (the **XM** bit) to 0.

See Also **SXM**

Example 4.14.63.1 `RXM`
Resets the sign extension mode to normal mode. Sets **XM** bit of **STAT** to 0.

4.14.64 SFLAG Set Memory Flag

Syntax

[label]	***name***	*dest*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	SFLAG	{*flagadrs*}	1	1	N/R	8a

Execution memory flag bit at {*flagadrs*} data memory location ⇐ 1
PC ⇐ **PC** + 1

Flags Affected None

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SFLAG *{flagadrs}*	1	0	0	1	1	1	0	1	0	1	*flagadrs*						

Description Set flag at addressed memory location. {*flagadrs*} includes two groups of memory flag adrresses: global flags, which are the first 64 words in RAM; and relative flags, which are 64 locations relative to the page register (**R6**). Flag address {*flagadrs*} only accesses the 17th bit.

See Also **RFLAG**, **STAG**, **RTAG**

Example 4.14.64.1 `SFLAG *R6+0x12`
Sets the flag bit of the RAM word addressed by **R6** plus 0x0002. Note that **R6** contains a byte address and 0x0002 is interpreted as a word offset.

4.14.65 SFM Set Fractional Mode

Syntax

[label]	***name***	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	SFM	1	1	N/R	9d

Execution **STAT.FM** ⇐ 1
PC ⇐ **PC** + 1

Flags Affected None

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RXM	1	1	1	1	1	1	1	1	0	1	1	0	0	0	0	0	0

Description Sets bit 3 (the **FM** bit) in status register (**STAT**) to 1. Enable multiplier shift mode for signed fractional arithmetic.

Example 4.14.65.1 `SFM`
Set fractional mode. Set **FM** bit of STAT to 1.

4.14.66 SHL Shift Left

Syntax

[label]	***name***	*dest [, mod]*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	SHL	**A***n*[**~**] [**,** *next A*]	1	1	n_R+3	3

Execution [premodify **AP** if *mod* specified]
PH, **PL** ⇐ *src* << **SV**
PC ⇐ **PC** + 1

Flags Affected **OF**, **SF**, **ZF**, **CF** are set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SHL A*n*[**~**] [**,** *next A*]	1	1	1	0	0	*next A*		**A***n*		1	1	1	1	1	0	**A~**	0

Description Premodify the accumulator pointer if specified. Shift accumulator word left n_{SV} bits (as specified by the **SV** register) into a 32-bit result. This result is zero-filled or sign-extended on the left (based on the setting of the extended sign mode (**XM**) bit in the status register). The upper 16 bits are latched into the **PH** register. Accumulator content is not changed. The lower 16-bit value, **PL**, is discarded. The **SHL** instruction can be used with a **RPT** instruction, but without much advantage since the instruction does not write back into the accumulator. Use **SHLAC** for this purpose.

See Also **SHLS**

Example 4.14.66.1
```
SHL A0, ++A
```
Preincrement accumulator pointer **AP0**. Shift accumulator word **A0** to the left by **SV** bits. Accumulator content is not changed. **PH** contains the upper 16 bits of the shifted result.

4.14.67 SHLAC Shift Left Accumulator

Syntax

[label]	***name***	*dest, src* [, *mod*]	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	SHLAC	**A***n*[~]**, A***n*[~] [**,** *next A*]	1	1	n_R+3	3

Execution [premodify **AP** if *mod* specified]
dest ⇐ *src* << 1
PC ⇐ **PC** + 1

Flags Affected **OF**, **SF**, **ZF**, **CF** are set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SHLAC A*n*[~]**, A***n*[~] [**,** *next A*]	1	1	1	0	0	*next A*		**A***n*		0	0	1	1	0	0	**A~**	**~A**

Description Premodify accumulator pointer if specified. Shift source accumulator *src* (or its offset) left by one bit and store the result in the destination accumulator (or its offset). LSB of result is set to zero.

Example 4.14.67.1 `SHLAC A1, A1`
Shift accumulator **A1** by one bit to the left.

Example 4.14.67.2 `SHLAC A1~, A1, --A`
Predecrement accumulator pointer **AP1** by 1. Shift the newly pointed accumulator **A1** by one bit to the left, store the result in accumulator **A1~**.

4.14.68 SHLACS Shift Left Accumulator String Individually

Syntax

[label]	***name***	*dest, src*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	SHLACS	**A***n*[~], **A***n*[~]	n_S+2	1	n_R+2	3

Execution

dest ⇐ *src* << 1
PC ⇐ **PC** + 1

Flags Affected **OF**, **SF**, **ZF**, **CF** are set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SHLACS A*n*[~], **A***n*[~]	1	1	1	0	0	1	1	**A***n*		0	0	1	1	0	0	**A~**	**~A**

Description Shift the source accumulator string *src* (or its offset) left one bit and store the result in destination accumulator string (or its offset). Each accumulator is shifted individually. The shifted bit is propagated through consecutive accumulators in the string.

Example 4.14.68.1

```
SHLACS A1~, A1
```

Shift accumulator string **A1** one bit to the left, store the result in accumulator string **A1~**. Note that this instruction alters the content of all accumulators in the string.

4.14.69 SHLAPL Shift Left with Accumulate

Syntax

[label]	***name***	*dest*, *src* [, *mod*]	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	SHLAPL	**A***n***,** {*adrs*}	Table 4–46		Table 4–46	1b
	SHLAPL	**A***n*[**~**]**, A***n*[**~**] [**,** *next A*]	1	1	n_R+3	3

Execution

[premodify **AP** if *mod* specified]
PH, **PL** ⇐ *src* << **SV**
dest ⇐ *dest* + **PL**
PC ⇐ **PC** + 1

Flags Affected

OF, **SF**, **ZF**, **CF** are set accordingly
src is {*adrs*}: **TAG** bit is set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SHLAPL A*n***,** {*adrs*}	0	1	1	1	1	0	0	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
SHLAPL A*n*[**~**]**, A***n*[**~**] [**,** *next A*]	1	1	1	0	0	*next A*		**A***n*		1	1	1	0	1	0	**A~**	**~A**

Description

Premodify the accumulator pointer if specified. Shift accumulator word or data memory word pointed by {*adrs*} to left n_{SV} bits (as specified by the **SV** register) into a 32-bit result. This result is zero-filled on the right and either zero-filled or sign-extended on the left (based on the setting of the extended sign mode (**XM**) bit in the status register). The upper 16 bits are latched into the product high (**PH**) register. The lower 16 bits of the result [product low (**PL**) register] is added to the destination accumulator (or its offset). This instruction propagates the shifted bits to the next accumulator.

Syntax	Description
SHLAPL A*n***,** {*adrs*}	Shift data memory word left, add **PL** to **A***n*
SHLAPL A*n*[**~**]**, A***n*[**~**] [**,** *next A*]	Shift **A***n*[**~**] left, add **PL** to **A***n*[**~**]

See Also **SHLAPLS, SHLTPL , SHLTPLS, SHLSPL, SHLSPLS**

Example 4.14.69.1 `SHLAPL A0, *R4++R5`

Shift the word pointed by the byte address stored in **R4** by n_{SV} bits to the left, add the shifted value (**PL**) with accumulator **A0**, store the result in accumulator **A0**. Add **R5** to **R4** and store result in **R4**. **PH** holds the upper 16 bits of the shift.

Example 4.14.69.2 `SHLAPL A2, *R1++`

Shift the word pointed by the byte address stored in **R1** by n_{SV} bits to the left, add the shifted value (**PL**) with the accumulator (**A2**), and store the result in accumulator **A2**. Increment **R1** (by 2) . **PH** holds the upper 16 bits of the shift.

Example 4.14.69.3 `SHLAPL A1, A1, ++A`

Preincrement accumulator pointer **AP1**. Shift the accumulator **A1** by n_{SV} bits to the left, add the shifted value (**PL**) to the accumulator and store the result in accumulator (**A1**). After execution **PH** contains the upper 16 bits of the 32-bit shift.

4.14.70 SHLAPLS Shift Left String With Accumulate

Syntax

[label]	***name***	*dest*, *src*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	SHLAPLS	**A***n***,** {*adrs*}	Table 4–46		Table 4–46	1b
	SHLAPLS	**A***n*[~]**, A***n*[~]	n_S+3	1	n_R+3	3

Execution

PH, **PL** ⇐ *src* << **SV**
dest ⇐ *dest* + **PL**
PC ⇐ **PC** + 1

Flags Affected

OF, **SF**, **ZF**, **CF** are set accordingly
src is {*adrs*}: **TAG** bit is set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SHLAPLS A*n*, {*adrs*}	0	1	1	1	1	0	1	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
SHLAPLS A*n*[~]**, A***n*[~]	1	1	1	0	0	1	1	**A***n*		1	1	1	0	1	0	**A~**	**~A**

Description

Shift accumulator string or data memory string pointed by {*adrs*} to left n_{SV} bits (as specified by the **SV** register). The result is zero-filled on the right and either zero-filled or sign-extended on the left (based on the setting of the extended sign mode (**XM**) bit in the status register). The upper 16 bits are latched into the product high (**PH**) register. The lower 16 bits of the result [product low (**PL**) register]) are added to the destination accumulator (or its offset). This instruction propagates the shifted bits to the next accumulators in the string.

Syntax	Description
SHLAPLS A*n***,** {*adrs*}	Shift data memory string left, add **PL** to **A***n*
SHLAPLS A*n*[~]**, A***n*[~]	Shift **A***n*[~] string left, addb **PL** to **A***n*[~]

See Also **SHLAPL , SHLTPL , SHLTPLS, SHLSPL, SHLSPLS**

Example 4.14.70.1 `SHLAPLS A0, *R4++R5`
Shift the string pointed by the byte address stored in **R4** by n_{SV} bits to the left, add the shifted value (**PL**) with accumulator string, and store the result in accumulator string **A0**. Add **R5** to **R4** and store result in **R4**. **PH** holds the upper 16 bits of the shift.

Example 4.14.70.2 `SHLAPLS A2, *R1++`
Shift the string pointed by the byte address stored in **R1** by n_{SV} bits to the left, add the shifted value (**PL**) with accumulator string, the accumulator, and store the result in accumulator string **A2**. Increment **R1** (by 2). **PH** holds the upper 16 bits of the shift.

Example 4.14.70.3 `SHLAPLS A1, A1`
Shift the accumulator string **A1** by n_{SV} bits to the left, add the shifted value (**PL**) to the accumulator and store the result in accumulator string **A1**. After execution **PH** contains the upper 16 bits of the 32-bit shift.

4.14.71 SHLS Shift Left Accumulator String to Product

Syntax

[label]	***name***	*dest*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	SHLS	**A***n*[~]	n_S+3	1	n_R+3	3

Execution **PH**, **PL** ⇐ *src* << **SV**
PC ⇐ **PC** + 1

Flags Affected **OF**, **SF**, **ZF**, **CF** are set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SHLS A*n*[~]	1	1	1	0	0	1	1	**A***n*		1	1	1	1	1	0	**A~**	0

Description Shift accumulator string value left n_{SV} bits (as specified by the **SV** register) into a ((n_S + 2) x 16) -bit result. The result is zero-filled or sign-extended on the left (based on the setting of the extended sign mode (**XM**) bit in the status register). The upper 16 bits are latched into the **PH** register. Accumulator content is not changed. The lower 16-bit value is discarded. **SHLS** instruction can be used with **RPT** instructions, but the string length used will be n_S + 2.

See Also **SHLS**

Example 4.14.71.1
```
SHLS A0
```
Shift accumulator string **A0** to the left. Accumulator content is not changed. **PH** contains the upper 16 bits of the shifted result.

4.14.72 SHLSPL Shift Left With Subtract PL

Syntax

[label]	***name***	*dest*, *src* [, *mod*]	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	SHLSPL	**A***n***,** {*adrs*}	Table 4–46		Table 4–46	1b
	SHLSPL	**A***n*[~]**, A***n*[~] [**,** *next A*]	1	1	n_R+3	3

Execution

[premodify **AP** if *mod* specified]
PH, **PL** ⇐ *src* << **SV**
dest ⇐ *dest* – **PL**
PC ⇐ **PC** + 1

Flags Affected

OF, **SF**, **ZF**, **CF** are set accordingly
src is {*adrs*}: **TAG** bit is set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SHLSPL A*n***,** {*adrs*}	0	1	1	1	0	1	0	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
SHLSPL A*n*[~]**, A***n*[~] [**,** *next A*]	1	1	1	0	0	*next A*		**A***n*		1	1	1	0	0	0	**A~**	**~A**

Description

Premodify the accumulator pointer if specified. Shift accumulator or data memory value pointed by {*adrs*} to left n_{SV} bits (as specified by the **SV** register) into a 32-bit result. This result is zero-filled on the right and either zero-filled or sign-extended on the left (based on the setting of the extended sign mode (**XM**) bit in the status register). The upper 16 bits are latched into the product high (**PH**) register. The lower 16 bits of the result [product low (**PL**) register] is subtracted from the destination accumulator (or its offset). This instruction propagates the shifted bit to the next accumulator.

Syntax	Description
SHLSPL A*n***,** {*adrs*}	Shift data memory word left, substract **PL** from **A***n*
SHLSPL A*n*[~]**, A***n*[~] [**,** *next A*]	Shift **A***n*[~] left, substract **PL** to **A***n*[~]

See Also **SHLSPLS, SHLTPL, SHLTPLS, SHLAPL, SHLAPLS**

Example 4.14.72.1 `SHLSPL A0, *R4++R5`

Shift the word pointed by the byte address stored in **R4** by n_{SV} bits to the left, subtract the shifted (**PL**) from Accummulator **A0**, and store the result in accumulator **A0**. Add **R5** to **R4** and store result in **R4** **PH** holds the upper 16 bits of the shift.

Example 4.14.72.2 `SHLSPL A2, *R1++`

Shift the word pointed by the byte address stored in **R1** by n_{SV} bits to the left, subtract the shifted value (**PL**) from the accumulator **A2**, and store the result in accumulator **A2**. Increment **R1** (by 2). **PH** holds the upper 16 bits of the shift.

Example 4.14.72.3 `SHLSPL A1, A1, ++A`

Preincrement accumulator pointer **AP1**. Shift the accumulator **A1** by n_{SV} bits to the left, subtract **PL** from **A1**, and store result in accululator **A1**. After execution **PH** contains the upper 16 bits of the 32-bit shift.

4.14.73 SHLSPLS Shift Left String With Subtract PL

Syntax

[label]	***name***	*dest*, *src*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	SHLSPLS	**A***n***,** {*adrs*}	Table 4–46		Table 4–46	1b
	SHLSPLS	**A***n*[~]**, A***n*[~]	n_S+3	1	n_R+3	3

Execution

PH, **PL** ⇐ *src* << **SV**
dest ⇐ *dest* – **PL**
PC ⇐ **PC** + 1

Flags Affected

OF, **SF**, **ZF**, **CF** are set accordingly
src is {*adrs*}: **TAG** bit is set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SHLSPLS A*n***,** {*adrs*}	0	1	1	1	0	1	1	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
SHLSPLS A*n*[~]**, A***n*[~]	1	1	1	0	0	1	1	**A***n*		1	1	1	0	0	0	**A~**	**~A**

Description

Shift accumulator string or data memory string pointed by {*adrs*} to left n_{SV} bits (as specified by the **SV** register). This result is zero-filled on the right and either zero-filled or sign-extended on the left (based on the setting of the extended sign mode (**XM**) bit in the status register). The upper 16 bits are latched into the **PH** register. The lower 16 bits of the result **PL** are subtracted from the destination accumulator (or its offset). This instruction propagates the shifted bit to the next accumulator.

Syntax	Description
SHLSPLS A*n***,** {*adrs*}	Shift RAM string left, subtract **PL** from **A***n*
SHLSPLS A*n*[~]**, A***n*[~]	Shift **A***n*[~] string left, subtract **PL** from **A***n*[~]

See Also **SHLSPL , SHLTPL , SHLTPLS, SHLAPL, SHLAPLS**

Example 4.14.73.1 `SHLSPLS A0, *R4++R5`

Shift the string pointed by the byte address stored in **R4** by n_{SV} bits to the left, subtract the shifted value (**PL**) from the value in the accumulator string in **A0**, and store the result in accumulator string **A0**. Add **R5** to **R4** and store result in **R4**. After execution of the instruction, **PH** is copied to the next to the last accumulator of the string.

Example 4.14.73.2 `SHLSPLS A2, *R1++`

Shift the string pointed by the byte address stored in **R1** by n_{SV} bits to the left, subtract the shifted value (**PL**) from the value in the accumulator string in **A2**, and store the result in accumulator string **A2**. Increment **R1** (by 2). After execution of the instruction, **PH** is copied to the next to the last accumulator of the string.

Example 4.14.73.3 `SHLSPLS A1, A1`

Shift the accumulator string **A1** by n_{SV} bits to the left, subtract the lower 16-bits of shifted value (**PL**) from **A1**, and store the result in **A1**. After execution **PH** contains the upper 16 bits of the 32-bit shift.

4.14.74 SHLTPL Shift Left and Transfer PL to Accumulator

Syntax

[label]	***name***	*dest*, *src* [, *mod*]	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	SHLTPL	**A***n***,** {*adrs*}	Table 4–46		Table 4–46	1b
	SHLTPL	**A***n*[~]**, A***n*[~] [**,** *next A*]	1	1	n_R+3	3

Execution

[premodify **AP** if *mod* specified]
PH, **PL** ⇐ *src* << **SV**
dest ⇐ **PL**
PC ⇐ **PC** + 1

Flags Affected

OF, **SF**, **ZF**, **CF** are set accordingly
src is {*adrs*}: **TAG** bit is set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SHLTPL A*n***,** {*adrs*}	0	1	1	1	0	0	0	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
SHLTPL A*n*[~]**, A***n*[~] [**,** *next A*]	1	1	1	0	0	*next A*		**A***n*		1	1	0	1	0	0	**A~**	**~A**

Description

Premodify the accumulator pointer if specified. Shift accumulator or data memory value pointed by {*adrs*} to left n_{SV} bits (as specified by the **SV** register) into a 32-bit result. The result is zero-filled on the right and either zero-filled or sign-extended on the left (based on the setting of the extended sign mode (**XM**) bit in the status register). The upper 16 bits are latched into the **PH** register. The lower 16 bits of the result **PL** are transferred to the destination accumulator (or its offset). This instruction propagates the shifted bit into **PH**.

Syntax	Description
SHLTPL A*n,* {*adrs*}	Shift data memory word left, transfer **PL** to **A***n*
SHLTPL A*n*[~]**, A***n*[~] [**,** *next A*]	Premodify **AP***n* if *next A* specified. Shift **A***n*[~] left, transfer **PL** to **A***n*[~]

See Also **SHLTPLS, SHLAPL, SHLAPLS, SHLSPL, SHLSPLS**

Example 4.14.74.1 `SHLTPL A0, *R4++R5`

Shift the word pointed by the byte address stored in **R4** by n_{SV} bits to the left, and store the result in accumulator **A0**. Add **R5** to **R4** and store result in **R4** at each execution to get the next memory value. After execution **PH** contains the upper 16 bits of the 32-bit shift.

Example 4.14.74.2 `SHLTPL A2, *R1++`

Shift the value pointed by the byte address stored in **R1** by n_{SV} bits to the left, and store the result in accumulator **A0**. Increment **R1** (by 2) at each execution to get the next memory value. After execution **PH** contains the upper 16 bits of the 32-bit shift.

Example 4.14.74.3 `SHLTPL A1, A1, ++A`

Preincrement accumulator pointer **AP1**. Shift the accumulator **A1** by n_{SV} bits to the left. After execution **PH** contains the upper 16 bits of the 32-bit shift.

4.14.75 SHLTPLS Shift Left String and Transfer PL to Accumulator

Syntax

[label]	***name***	*dest*, *src*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	SHLTPLS	**A***n***,** {*adrs*}	Table 4–46		Table 4–46	1b
	SHLTPLS	**A***n*[~]**, A***n*[~]	n_S+3	1	n_R+3	3

Execution

PH, **PL** ⇐ *src* << **SV**
dest ⇐ **PL**
PC ⇐ **PC** + 1

Flags Affected

OF, **SF**, **ZF**, **CF** are set accordingly
src is {*adrs*}: **TAG** bit is set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SHLTPLS A*n***,** {*adrs*}	0	1	1	1	0	0	1	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
SHLTPLS A*n*[~]**, A***n*[~]	1	1	1	0	0	1	1	**A***n*		1	1	0	1	0	0	**A~**	**~A**

Description

Shift left accumulator string or data memory string pointed at by {*adrs*} by n_{SV} bits (as specified by the **SV** register). The result is zero-filled on the right and either zero-filled or sign-extended on the left (based on the setting of the Extended Sign Mode (**XM**) bit in the status register). The upper 16 bits are latched into the **PH** register. The result is transferred to the destination accumulator (or its offset). This instruction propagates the shifted bits to the next accumulator, including one accumulator past the string length (which receives the same data as **PH**).

Syntax	Description
SHLTPLS A*n***,** {*adrs*}	Shift data memory string left, transfer result to **A***n*
SHLTPLS A*n*[~]**, A***n*[~]	Shift **A***n*[~] string left, transfer result to **A***n*[~]

See Also **SHLTPL, SHLAPL, SHLAPLS, SHLSPL, SHLSPLS**

Example 4.14.75.1 `SHLTPLS A0, *R4++R5`

Shift the string pointed by the byte address stored in **R4** by n_{SV} bits to the left, and store the result in accumulator string **A0**. Add **R5** to **R4** and store result in **R4**. After execution of the instruction, **PH** is copied to the next to the last accumulator of the string.

Example 4.14.75.2 `SHLTPLS A2, *R1++`

Shift the string pointed by the byte address stored in **R1** by n_{SV} bits to the left, and store the result in accumulator string **A0**. Increment **R1** (by 2) at each execution to get the next memory value.

Example 4.14.75.3 `SHLTPLS A1, A1`

Shift the accumulator string **A1** by n_{SV} bits to the left.

4.14.76 SHRAC Shift Accumulator Right

Syntax

[label]	***name***	*dest, src,* [, *mod*]	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	SHRAC	**A***n*[**~**]**,** **A***n*[**~**] [**,** *next A*]	1	1	n_R+3	3

Execution [premodify **AP** if *mod* specified]
dest ⇐ *src* >> 1
PC ⇐ **PC** + 1

Flags Affected **OF**, **SF**, **ZF**, **CF** are set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SHRAC A*n*[**~**], **A***n*[~] [, *next a*]	1	1	1	0	0	*next A*		**A***n*		0	1	0	1	1	0	**A~**	**~A**

Description Premodify accumulator pointer if specified. Shift source accumulator *src* or its offset to right one bit and store the result into *dest* accumulator or its offset. MSB of result will be set according to extended sign mode (**XM**) bit in the status register.

Example 4.14.76.1 `SHRAC A1, A1`
Shift right one bit the accumulator **A1**.

Example 4.14.76.2 `SHRAC A1~, A1, ++A`
Preincrement by one accumulator pointer **AP1**. Shift right one bit the newly pointed accumulator **A1**, and store result to offset accumulator **A1~**.

4.14.77 SHRACS Shift Accumulator String Right

Syntax

[label]	***name***	*dest, src*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	SHRACS	**A***n*[~]**, A***n*[~]	n_S+3	1	n_R+3	3

Execution *dest* ⇐ *src* >> 1
PC ⇐ **PC** + 1

Flags Affected **OF**, **SF**, **ZF**, **CF** are set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SHRACS A*n*[~]**, A***n*[~]	1	1	1	0	0	1	1	**A***n*		0	1	0	1	1	0	**A~**	**~A**

Description Shift accumulator string right one bit and store the result into **A***n*[**~**] string. MSB of each accumulator in the result will be set according to extended sign mode (**XM**) bit in the status register. This instruction shifts each accumulator individually 1 bit to the right, so, shifts from one accumulator are not propagated to the next consecutive accumulator in the string.

See Also **SHRAC**, **SHL**, **SHLS**, **SHLAPL**, **SHLAPLS**, **SHLSPL**, **SHLSPLS**, **SHLTPL**, **SHLTPLS**.

Example 4.14.77.1
```
SHRACS A0, A0
```
Shift accumulator string **A0** 1 bit right individually.

Example 4.14.77.2
```
SHRACS A1, A1~
```
Shift accumulator string **A1~** individually, put result in accumulator string **A1**.

4.14.78 SOVM Set Overflow Mode

Syntax

[label]	***name***	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	SOVM	1	1	N/R	9d

Execution **STAT.OM** ⇐ 1
PC ⇐ **PC** + 1

Flags Affected None

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SOVM	1	1	1	1	1	1	1	1	0	1	1	0	1	0	0	0	0

Description Sets overflow mode in status register (**STAT**) bit 2 to 1. Enable ALU saturation output (DSP mode).

See Also **ROVM**

Example 4.14.78.1 `SOVM`
Set **OM** bit of **STAT** to 1. This is the mode DSP algorithms should use.

4.14.79 STAG Set Tag

Syntax

[label]	***name***	*dest*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	STAG	{*adrs*}	Table 4–46		Table 4–46	5

Execution memory tag bit at address adrs ⇐ 1
PC ⇐ **PC** + *w*

Flags Affected None

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
STAG {*adrs*}	1	1	0	1	0	1	1	0	0	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															

Description Sets the tag bit at the addressed memory location. All addressing modes are available. Note that this instruction accesses only the 17th bit of the RAM location. The argument, {*adrs*}, is interpreted as bytes. For odd RAM byte addresses, the least significant bit is ignored.

See Also **RTAG**, **RFLAG**, **SFLAG**

Example 4.14.79.1 `STAG *R2+R5`
Set **TAG** bit of the word in RAM byte address, **R2** + **R5**. **R2** and **R5** remain unchanged.

Example 4.14.79.2 `STAG *0x200 * 2`
Set **TAG** bit of RAM word 0x200 (RAM byte address 0x400).

Example 4.14.79.3 `STAG *0x401`
Set **TAG** bit of RAM word 0x200 (RAM byte address 0x400).

4.14.80 SUB Subtract

Syntax

[label]	***name***	*dest, src*, *src1*, [*next A*]]	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	SUB	**A***n*[~], **A***n*, {*adrs*} [, *next A*]	Table 4–46		Table 4–46	1a
	SUB	**A***n*[~], **A***n*[~], *imm16* [, *next A*]	2	2	N/R	2b
	SUB	**A***n*[~], **A***n*[~], **PH** [, *next A*]	1	1	n_R+3	3
	SUB	**A***n*[~], **A***n*, **A***n*~ [, *next A*]	1	1	n_R+3	3
	SUB	**A***n*[~], **A***n*~, **A***n* [, *next A*]	1	1	n_R+3	3
	SUB	**R***x*, *imm16*	2	2	N/R	4c
	SUB	**R***x*, **R5**	1	1	N/R	4d

Execution

[premodify **AP** if *mod* specified]
dest ⇐ *dest* – *src1* (for two operands)
dest ⇐ *src* – *src1* (for three operands)
PC ⇐ **PC** + *w*

Flags Affected

dest is **A***n*: **OF**, **SF**, **ZF**, **CF** are set accordingly
dest is **R***x*: **RCF**, **RZF** are set accordingly
src1 is {*adrs*}: **TAG** bit is set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SUB A*n*[~], **A***n*, {*adrs*} [, *next A*]	0	0	0	0	**~A**	*next A*		**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
SUB A*n*[~], **A***n*[~], *imm16* [, *next A*]	1	1	1	0	0	*next A*		**A***n*		0	1	0	0	0	1	**A~**	**~A**
SUB A*n*[~], **A***n*[~], **PH** [, *next A*]	1	1	1	0	0	*next A*		**A***n*		0	1	1	0	0	0	**A~**	**~A**
SUB A*n*[~], **A***n*, **A***n*~ [, *next A*]	1	1	1	0	0	*next A*		**A***n*		0	0	1	0	0	0	0	**~A**
SUB A*n*[~], **A***n*~, **A***n* [, *next A*]	1	1	1	0	0	*next A*		**A***n*		0	0	1	0	0	0	1	**~A**
SUB R*x*, *imm16*	1	1	1	1	1	1	1	0	0	0	0	1	**R***x*			0	0
SUB R*x*, **R5**	1	1	1	1	1	1	1	0	0	1	0	1	**R***x*			0	0

Description

Subtract value of *src* from value of *dest* and store result in *dest*. If three operands are specified, then subtract value of *src1* from value of *src* (i.e., *src-src1*) and store result in *dest* string. Premodification of accumulator pointers is allowed with some operand types. Note that subtraction is performed in 2's complement and therefore the **CF** (carry flag) may get set even when subtracting a smaller value from a larger value.

Syntax	Description
SUB A*n*[**~**]**, A***n***,** {*adrs*} [, *next A*]	Subtract effective data memory word from **A***n*[**~**], store result in **A***n*
SUB A*n*[**~**]**, A***n*[**~**]**,** *imm16* [, *next A*]	Subtract immediate word from **A***n*[**~**], store result in **A***n*[**~**]
SUB A*n*[**~**]**, A***n*[**~**]**, PH** [**,** *next A*]	Subtract Product High (**PH**) register from **A***n*[**~**], store result in **A***n*[**~**]
SUB A*n*[**~**]**, A***n***, A***n***~** [**,** *next A*]	Subtract **A***n***~** word from **A***n* word, store result in **A***n*[**~**]
SUB A*n*[**~**]**, A***n***~, A***n* [**,** *next A*]	Subtract **A***n* word from **A***n***~** word, store result in **A***n*[**~**]
SUB R*x***,** *imm16*	Subtract immediate word from **R***x*
SUB R*x***, R5**	Subtract **R5** from **R***x*

See Also **SUBB, SUBS, ADD, ADDB, ADDS**

Example 4.14.80.1 `SUB A1, A1, 74`
Subtract 74 (decimal) immediate from accumulator **A1**, put result in accumulator **A1**.

Example 4.14.80.2 `SUB A0, A0, 2, ++A`
Pre–increment pointer **AP0**, subtract 2 from new accumulator **A0**, put result in accumulator **A0**.

Example 4.14.80.3 `SUB A1, A1~, A1`
Subtract accumulator **A1** from accumulator **A1~**, put result in accumulator **A1**.

Example 4.14.80.4 `SUB A1, A1, A1~, --A`
Pre–decrement **AP1**. Subtract accumulator **A1~** from accumulator **A1**, put result in accumulator **A1**.

Example 4.14.80.5 `SUB A3~, A3, *R4–`
Subtract word at address in **R4** from **A3**, store result in **A3~**, decrement value in **R4** by 2 (word mode) after the subtraction.

Example 4.14.80.6 `SUB R3, R5`
Subtract **R5** from **R3**, put result in **R3**.

4.14.81 SUBB Subtract Byte

Syntax

[label]	***name***	*dest, src*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	SUBB	**A***n***,** *imm8*	1	1	N/R	2a
	SUBB	**R***x***,** *imm8*	1	1	N/R	4b

Execution *dest* ⇐ *dest* – *imm8*
PC ⇐ **PC** + 1

Flags Affected *dest* is **A***n*: **OF**, **SF**, **ZF**, **CF** are set accordingly
dest is **R***x*: **RCF**, **RZF** are set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SUBB A*n imm8*	1	0	1	0	0	1	0	**A***n*		*imm8*							
SUBB R*x*, *imm8*	1	0	1	1	0	1	*k4*	*k3*	*k2*	*k7*	*k6*	*k5*	**R***x*			*k1*	*k0*

Description Subtract value of *src* byte from value of *dest* byte and store result in *dest*. Note that subtraction is performed in 2's complement and therefore the **CF** (carry flag) may get set even when subtracting a smaller value from a larger value.

Syntax	**Description**
SUBB A*n***,** *imm8*	Subtract immediate byte from **A***n*
SUBB R*x***,** *imm8*	Subtract immediate byte from **R***x*

Example 4.14.81.1 `SUBB A2, 0x45`
Subtract 0x45 from accumulator **A2** byte.

Example 4.14.81.2 `SUBB R3, 0xF2`
Subtract 0xF2 from register **R3** byte.

4.14.82 SUBS Subtract Accumulataor String

Syntax

[label]	***name***	*dest, src, src1*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	SUBS	**A***n*[~], **A***n*, {*adrs*}	Table 4–46		Table 4–46	1a
	SUBS	**A***n*[~], **A***n*[~], *pma16*	n_S+4	2	N/R	32b
	SUBS	**A***n*[~], **A***n*, **A***n*~	n_S+2	1	n_R+2	3
	SUBS	**A***n*[~], **A***n*~, **A***n*	n_S+2	1	n_R+2	3
	SUBS†	**A***n*[~], **A***n*[~], **PH**	1	1	1	3

† This instruction ignores the string count, executing only once but maintains the **CF** and **ZF** status of the previous multiply or shift operation as if the sequence was a single string. This instruction should immediately follow one of the following class 1b instructions: MOVAPH, MULAPL, MULSPL, SHLTPL, SHLSPL, and SHLAPL. An interrupt can occur between one of these instructions and this instruction. An interrupt may cause an incorrect result. Also, single stepping is not allowed for this instruction. **A***n* in this instruction should be the same as **A***n* in one of the listed class 1b instruction. Offsets are allowed. See Section 4.8 for detail.

Execution

[premodify **AP** if *mod* specified]
dest ⇐ *dest* – *src* (for two operands)
dest ⇐ *src* – *src1* (for three operands)
PC ⇐ **PC** + *w*

Flags Affected

dest is **A***n*: **OF**, **SF**, **ZF**, **CF** are set accordingly
src1 is {*adrs*}: **TAG** bit is set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SUBS A*n*[~], **A***n*, {*adrs*}	0	0	0	1	**~A**	1	1	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
SUBS A*n*[~], **A***n*[~], *pma16*	1	1	1	0	0	1	1	**A***n*		0	1	0	0	1	1	**A~**	**~A**
SUBS A*n*[~], **A***n*, **A***n*~	1	1	1	0	0	1	1	**A***n*		0	0	1	0	0	0	0	**~A**
SUBS A*n*[~], **A***n*~, **A***n*	1	1	1	0	0	1	1	**A***n*		0	0	1	0	0	0	1	**~A**
SUBS A*n*[~], **A***n*[~], **PH**	1	1	1	0	0	1	1	**A***n*		0	1	1	0	0	0	**A~**	**~A**

Description

Subtract the value of the *src* string from value of the *dest* string and store the result in the *dest* string. If three operands are specified, then subtract value of *src1* string from value of *src* string (i.e., *src*–*src1*) and store result in *dest* string. Note that, substraction is performed in 2's complement and therefore the **CF** (carry flag) may get set even when subtracting a smaller value from a large value.

Syntax	Description
SUBS A*n*[~], **A***n*, {*adrs*}	Subtract data memory string from **A***n* string, store result in **A***n*[~] string
SUBS A*n*[~], **A***n*[~], *pma16*	Subtract program memory string from **A***n*[~] string, store result in **A***n*[~] string
SUBS A*n*[~], **A***n*, **A***n*~	Subtract **A***n*~ string from **A***n* string, store result in **A***n*[~] string
SUBS A*n*[~], **A***n*~, **A***n*	Subtract **A***n* string from **A***n*~ string, store result in **A***n*[~] string
SUBS A*n*[~], **A***n*[~], **PH**	Subtract product high (**PH**) register from **A***n*[~] string mode. This instruction ignores the string count, executing only once but maintains the **CF** and **ZF** status of the previous multiply or shift operation as if the sequence was a single string. Word alignment with **PH** is maintained, i.e., **PH** is subtracted from the second word of the string. Also, only the second word is copied to the destination string.

Example 4.14.82.1 `SUBS A0, A0~, *R2++`
Subtract data memory string beginning at address in **R2** from accumulator string **A0~**, put result in accumulator string **A0** then increment **R2** by 2.

Example 4.14.82.2 `SUBS A1~, A1, 0x1220`
Subtract program memory string at address 0x1220 from accumulator string **A1**, put result in accumulator string **A1~**.

Example 4.14.82.3 `SUBS A2, A2, A2~`
Subtract accumulator string **A2~** from accumulator string **A2**, put result in accumulator string **A2**.

Example 4.14.82.4 `SUBS A2, A2~, A2`
Subtract accumulator string **A2** from accumulator string **A2~**, put result in accumulator string **A2**.

Example 4.14.82.5 `SUBS A3~, A3~, PH`
Subtract **PH** from accumulator string **A3~**, put result in accumulator string **A3**. This instruction ignores the string count.

4.14.83 SXM Set Extended Sign Mode

Syntax

[label]	***name***	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	SXM	1	1	N/R	9d

Execution **STAT.XM** ⇐ 1
PC ⇐ **PC** + 1

Flags Affected None

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SXM	1	1	1	1	1	1	1	1	0	1	0	1	0	0	0	0	0

Description Sets extended sign mode status register (**STAT**) bit 0 to 1.

See Also **RXM**

Example 4.14.83.1 SXM
Set **XM** bit of **STAT** to 1. Now all arithematic operation will be in sign extention mode.

4.14.84 VCALL Vectored Call

Syntax

[label]	***name***	*dest*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	VCALL	*vector8*	2	1	N/R	7a

Execution

Push **PC** + 1
PC ⇐ *(0x7F00 + *vector8*)
R7 ⇐ **R7** + 2

Flags Affected

None

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VCALL *vector8*	1	1	1	1	1	1	1	0	1	*vector8*							

Description

Unconditional vectored call (Macro call). Push next address onto stack, load **PC** with the content of the address obtained by adding *vector8* to 0x7F00. The execution of the instruction continues from the new **PC** location. **RET** instruction is used to return from **VCALL**. **RET** cannot immediately follow **VCALL**. **IRET** can be used instead of **RET** and **IRET** can immidiately follow **VCALL**. **VCALL** is used to call frequently used routines and takes 1 word.

See Also

RET, IRET, CALL, C*cc*

Example 4.14.84.1 `VCALL 0x7F02`

Loads **PC** value with the program memory address stored in program memory location 0x7F02.

4.14.85 XOR Logical XOR

Syntax

[label]	***name***	*dest, src, src1* [, *mod*]	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	XOR	**A***n***,** {*adrs*}	Table 4–46		Table 4–46	1a
	XOR	**A***n*[**~**]**, A***n*[**~**]**,** *imm16* [**,** *next A*]	2	2	N/R	2b
	XOR	**A***n*[**~**]**, A***n***~, A***n* [**,** *next A*]	1	1	n_R+3	3
	XOR	**TF***n***,** {*flagadrs*}	1	1	N/R	8a
	XOR	**TF***n***,** {*cc*} [**, R***x*]	1	1	n_R+3	8b

Execution

[premodify **AP** if *mod* specified]
dest ⇐ *dest* XOR *src* (for two operands)
dest ⇐ *src1* XOR *src* (for three operands)
PC ⇐ **PC** + *w*

Flags Affected

dest is **A***n*: **OF**, **SF**, **ZF**, **CF** are set accordingly
dest is **TF***n*: **TF***n* bits in **STAT** register are set accordingly
src is {*adrs*}: **TAG** bit is set accordingly
src is {flag*adrs*}: **TAG** bit is set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
XOR A*n***,** {*adrs*}	0	1	0	0	1	0	0	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
XOR A*n*[**~**]**, A***n*[**~**]**,** *imm16* [**,** *next A*]	1	1	1	0	0	*next A*		**A***n*		1	1	0	0	0	1	**A~**	**~A**
XOR A*n*[**~**]**, A***n***~, A***n* [**,** *next A*]	1	1	1	0	0	*next A*		**A***n*		0	1	0	0	0	0	**A~**	**~A**
XOR TF*n***,** {*flagadrs*}	1	0	0	1	1	*fig*	*Not*	1	1	0	*flagadrs*						
XOR TF*n***,** {*cc*} [**, R***x*]	1	0	0	1	0	*fig*	*Not*	*cc*				**R***x*				1	1

Description

Bitwise logical XOR of *src* and *dest*. Result is stored in *dest*. If three operands are specified, then logical XOR *src* and *src1*, store the result in *dest*. Premodification of accumulator pointers is allowed with some operand types.

Syntax	**Description**
XOR A*n***,** {*adrs*}	XOR RAM word to **A***n*
XOR A*n*[**~**]**, A***n*[**~**]**,** *imm16* [**,** *next A*]	XOR immediate word to **A***n*[**~**], store result in **A***n*[**~**]
XOR A*n*[**~**]**, A***n***~, A***n* [, *next A*]	XOR **A***n* word to **A***n***~** word, store result in **A***n*[**~**]
XOR TF*n***,** {*flagadrs*}	XOR **TF***n* (either **TF1** or **TF2**) with memory tag, store result in **TF***n* bit in **STAT**
XOR TF*n***,** {*cc*} [**, R***x*]	XOR test condition with **TF***n* (either **TF1** or **TF2**) bit in **STAT** register. **R***x* must be provided if *cc* is one of {**RZP**, **RNZP**, **RLZP**, **RNLZP**} to check if the selected **R***x* is zero or negative. **R***x* should not be provided for other conditionals.

See Also **XORB, XORS, AND, ANDS, OR, ORS, ORB, NOTAC, NOTACS**

Example 4.14.85.1 `XOR A1, A1, 0x13FF`
XOR immediate value 0x13FF to **A1** and store result in **A1**.

Example 4.14.85.2 `XOR A0, A0, 2, ++A`
Pre–increment pointer **AP0**, then XOR immediate value 2 to new **A0** and store result in **A0**.

Example 4.14.85.3 `XOR A1, A1~, A1`
XOR accumulator **A1** to accumulator **A1~**, put result in accumulator **A1**.

Example 4.14.85.4 `XOR A3, *R4–`
XOR word at address in **R4** to accumulator **A3**, decrement value in **R4** by 2 (word mode) after the operation.

Example 4.14.85.5 `XOR A2, A2~, *R2+R5, ––A`
Pre–decrement pointer **AP2**. XOR word at effective address **R2+R5** to new accumulator **A2~**, put result in accumulator **A2**. Value of **R2** is not modified.

Example 4.14.85.6 `XOR TF1, *0x21`
XOR **TF1** with the flag at global address 0x21 and store result in **TF1** in **STAT**.

Example 4.14.85.7 `XOR TF2, *R6+0x21`
XOR **TF2** with the flag at effective address **R6**+0x21 and store result in **TF2**.

Example 4.14.85.8 `XOR TF1, CF`
XOR **TF1** with the condition code **CF** (Carry Flag) and store result in **TF1**.

Example 4.14.85.9 `XOR TF1, RZP, R3`
XOR **TF1** with the condition code **RZP** (**R***x*=0 flag) for **R3**, and store result in **TF1**. If the content of **R3** is zero then **RZP** condition becomes true, otherwise false.

4.14.86 XORB Logical XOR Byte

Syntax

[label]	***name***	*dest, src*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	XORB	**A***n***,** *imm8*	1	1	N/R	2a

Execution **A***n* ⇐ **A***n* XOR *imm8* (for two operands)
PC ⇐ **PC** + 1

Flags Affected *dest* is **A***n*: **OF**, **SF**, **ZF**, **CF** are set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
XORB A*n***,** *imm8*	0	0	1	0	1	1	0	**A***n*		*imm8*							

Description Bitwise logical XOR lower 8 bits of **A***n* and *dest* byte. Result is stored in accumulator **A***n*. Upper 8 bits of accumulator **A***n* is not affected.

See Also **XOR, XORS, AND, ANDS, OR, ORS, ORB, NOTAC, NOTACS**

Example 4.14.86.1
```
XORB A2, 0x45
```
XOR 0x45 to accumulator **A2** (byte mode). Upper 8 bits of **A2** is unchanged.

4.14.87 XORS Logical XOR String

Syntax

[label]	***name***	*dest, src* [*, src1*]	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	XORS	**A***n***,** {*adrs*}	Table 4–46		Table 4–46	1b
	XORS	**A***n*[~]**, A***n*[~]**,** *pma16*	n_S+4	2	N/R	2b
	XORS	**A***n*[~]**, A***n***~, A***n*	n_S+3	1	n_R+3	3

Execution

dest ⇐ *dest* XOR *src* (for two operands)
dest ⇐ *src1* XOR *src* (for three operands)
PC ⇐ **PC** + *w*

Flags Affected

dest is **A***n*: **OF**, **SF**, **ZF**, **CF** are set accordingly
src is {*adrs*}: **TAG** bit is set accordingly

Opcode

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
XORS A*n***,** {*adrs*}	0	1	0	0	1	0	1	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
XORS A*n*[~]**, A***n*[~]**,** *pma16*	1	1	1	0	0	1	1	**A***n*		1	1	0	0	0	1	**A~**	**~A**
XORS A*n*[~]**, A***n***~, A***n*	1	1	1	0	0	1	1	**A***n*		0	1	0	0	0	0	**A~**	**~A**

Description

Bitwise XOR of *src* string and *dest* string. Result is stored in *dest* string. If three operands are specified, then logical XOR *src* string and *src1* string, store result in *dest* string.

Syntax	Description
XORS A*n***,** {*adrs*}	XOR data memory string to **A***n* string
XORS A*n*[~]**, A***n*[~]**,** *pma16*	XOR program memory string to **A***n*[~] string, store result in **A***n*[~] string
XORS A*n*[~]**, A***n***~, A***n*	XOR **A***n* string to **A***n***~** string, store result in **A***n*[~] string

See Also **XOR, XORB, AND, ANDS, OR, ORS, ORB, NOTAC, NOTACS**

Example 4.14.87.1
```
XORS A0, A0~, *R2
```
XOR data memory string beginning at address in **R2** to accumulator string **A0~**, put result in accumulator string **A0**.

Example 4.14.87.2
```
XORS A3~, A3, *R1++R5
```
XOR data memory string beginning at address in **R1** to accumulator string **A3**, put result in accumulator string **A3~**. Add value in **R5** to the value in **R1** and store result in **R1**.

Example 4.14.87.3
```
XORS A1~, A1, 0x100 * 2
```
XOR program memory string beginning at word address 0x0100 to accumulator string **A1**, put result in accumulator string **A1~**.

Example 4.14.87.4
```
XORS A2, A2~, A2
```
XOR accumulator string **A2** with accumulator string **A2~** string, put result in accumulator string **A2**.

4.14.88 ZAC Zero Accumulator

Syntax

[label]	***name***	*dest* [*, mod*]	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	ZAC	**A***n*[**~**] [**,** *next A*]	1	1	n_R+3	3

Execution [premodify **AP** if *mod* specified]
dest ⇐ 0
PC ⇐ **PC** + 1

Flags Affected **ZF** = 1

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ZAC A*n*[~] [**,** *next A*]	1	1	1	0	0	*next A*		**A***n*		0	0	0	1	1	0	0	**~A**

Description Zero the specified accumulator. Preincrement or predecrement accumulator pointer **AP***n*, if specified.

See Also **ZACS**

Example 4.14.88.1 `ZAC A2`
Reset the content of accumulator **A0** to zero.

Example 4.14.88.2 `ZAC A1~, ++A`
Preincrement **AP1** by 1. Reset the content of new accumulator **A1~** to zero.

4.14.89 ZACS Zero Accumulator String

Syntax

[label]	***name***	*dest*	**Clock**, *clk*	**Word**, *w*	**With RPT**, *clk*	**Class**
	ZAC	**A***n*	n_S+3	1	n_R+3	3

Execution *dest* ⇐ 0
PC ⇐ **PC** + 1

Flags Affected **ZF** = 1

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ZACS A*n*[~]	1	1	1	0	0	1	1	**A***n*		0	0	0	1	1	0	0	**~A**

Description Zero the specified accumulator string.

See Also **ZAC**

Example 4.14.89.1
```
ZACS A1~
```
Reset the content of offset accumulator string **A1~** to zero.

Example 4.14.89.2
```
MOV STR, 32-2
ZACS A0
```
Reset the content of all accumulators to zero. It does not matter which accumulator **AP0** is pointing at since all the accumulators are zeroed.

4.15 Instruction Set Encoding

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD A*n*[~], **A***n*, {*adrs*} [, *next A*]	**1**	**1**	**1**	**0**	**~A**	*next A*		**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
ADD A*n*[~], **A***n*[~], *imm16* [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**0**	**0**	**0**	**0**	**0**	**1**	**A~**	**~A**
	x	*imm16*															
ADD A*n*[~], **A***n*[~], **PH** [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**0**	**1**	**1**	**0**	**1**	**0**	**A~**	**~A**
ADD A*n*[~], **A***n*~, **A***n* [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**0**	**0**	**1**	**0**	**1**	**0**	**A~**	**~A**
ADD R*x*, *imm16*	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**R***x*			**0**	**0**
	x	*imm16*															
ADD R*x*, **R5**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**R***x*			**0**	**0**
ADD AP*n*, *imm5*	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**AP***n*		**0**	**1**	**0**	*imm5*				
ADDB A*n*, *imm5*	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**A***n*		*imm8*							
ADDB R*x*, *imm8*	**1**	**0**	**1**	**1**	**0**	**0**	*k4*	*k3*	*k2*	*k7*	*k6*	*k5*	**R***x*			*k1*	*k0*
ADDS A*n*[~], A*n*, {*adrs*}	**0**	**0**	**0**	**0**	**~A**	**1**	**1**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
ADDS A*n*[~], **A***n*[~], *pma16*	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**0**	**0**	**0**	**0**	**0**	**1**	**A~**	**~A**
	x	*pma16*															
ADDS A*n*[~], **A***n*~, **A***n*	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**0**	**0**	**1**	**0**	**1**	**0**	**A~**	**~A**
ADDS A*n*[~], **A***n*[~], **PH**	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**0**	**1**	**1**	**0**	**1**	**0**	**A~**	**~A**
AND A*n*, {*adrs*}	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
AND A*n*[~], **A***n*[~], *imm16* [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**1**	**0**	**1**	**0**	**0**	**1**	**A~**	**~A**
	x	*imm16*															
AND A*n*[~], **A***n*~, **A***n* [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**0**	**1**	**0**	**1**	**0**	**0**	**A~**	**~A**
AND TF*n*, {*flagadrs*}	**1**	**0**	**0**	**1**	**1**	*flg*	*Not*	**1**	**0**	**0**	*flagadrs*						
AND TF*n*, {*cc*} [, **R***x*]	**1**	**0**	**0**	**1**	**0**	*flg*	*Not*	*cc*					**R***x*			**1**	**0**
ANDB A*n*, *imm8*	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**A***n*		*imm8*							
ANDS A*n*, {*adrs*}	**0**	**1**	**0**	**0**	**0**	**1**	**1**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
ANDS A*n*[~], **A***n*[~], *pma16*	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**1**	**0**	**1**	**0**	**0**	**1**	**A~**	**~A**
ANDS A*n*[~], **A***n*~, **A***n*	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**0**	**1**	**0**	**1**	**0**	**0**	**A~**	**~A**
BEGLOOP	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
CALL *pma16*	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**0**
	x	*pma16*															
CALL *A*n*	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**A***n*		**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
C*cc* *pma16*	**1**	**0**	**0**	**0**	**0**	**1**	*Not*	*cc*					**0**	**0**	**0**	**0**	**0**
	x	*pma16*															

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CMP A*n*, {*adrs*}	**0**	**1**	**0**	**1**	**1**	**0**	**0**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
CMP A*n*[~], *imm16* [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**0**	**1**	**1**	**0**	**0**	**1**	**A~**	**~A**
	x	*imm16*															
CMP A*n*, **A***n*~ [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
CMP A*n*~, **A***n* [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**
CMP R*x*, *imm16*	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	**R***x*			**0**	**0**
	x	*imm16*															
CMP R*x*, **R5**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**1**	**R***x*			**0**	**0**
CMPB A*n*, *imm8*	**1**	**0**	**1**	**0**	**0**	**1**	**1**	**A***n*		*imm8*							
CMPB R*x*, *imm8*	**1**	**0**	**1**	**1**	**1**	**1**	*k4*	*k3*	*k2*	*k7*	*k6*	*k5*	**R***x*			*k1*	*k0*
CMPS A*n*, {*adrs*}	**0**	**1**	**0**	**1**	**1**	**0**	**1**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
CMPS A*n*[~], *pma16*	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**0**	**1**	**1**	**0**	**0**	**1**	**A~**	**0**
	x	*pma16*															
CMPS A*n*, **A***n*~	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
CMPS A*n*~, **A***n*	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**
COR A*n*, ***R***x*	**1**	**1**	**1**	**0**	**1**	**0**	**0**	**A***n*		**1**	**1**	**0**	**R***x*			**1**	**1**
CORK A*n*, ***R***x*	**1**	**1**	**1**	**0**	**1**	**0**	**0**	**A***n*		**1**	**0**	**0**	**R***x*			**1**	**1**
ENDLOOP *n*	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	*n*
EXTSGN A*n*[~] [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**0**	**1**	**1**	**1**	**1**	**0**	**0**	**~A**
EXTSGNS A*n*[~]	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**0**	**1**	**1**	**1**	**1**	**0**	**0**	**A~**
FIR A*n*, ***R***x*	**1**	**1**	**1**	**0**	**1**	**0**	**0**	**A***n*		**0**	**1**	**0**	**R***x*			**1**	**1**
FIRK A*n*, ***R***x*	**1**	**1**	**1**	**0**	**1**	**0**	**0**	**A***n*		**0**	**0**	**0**	**R***x*			**1**	**1**
IDLE	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**
IN {*adrs*}, *port4*	**1**	**1**	**0**	**0**	**0**	*port4*				*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
IN A*n*[~], *port6*	**1**	**1**	**1**	**0**	**1**	**1**	**0**	**A***n*		*port6*						**0**	**~A**
INS A*n*[~], *port6*	**1**	**1**	**1**	**0**	**1**	**1**	**1**	**A***n*		*port6*						**0**	**~A**
INTD	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**0**
INTE	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
IRET	**1**	**1**	**0**	**1**	**1**	**1**	**1**	**0**	**1**	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**0**
JMP *pma16*	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**0**
	x	*pma16*															
JMP *pma16*, **R***x*++	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**1**	**R***x*			**0**	**1**
	x	*pma16*															

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JMP *pma16*, **R***x***––**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**1**	**R***x*			**1**	**0**
	x	*pma16*															
JMP *pma16*, **R***x***++R5**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**1**	**R***x*			**1**	**1**
	x	*pma16*															
JMP *A*n*	**1**	**0**	**0**	**0**	**1**	**0**	**0**	**A***n*		**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
J*cc* *pma16*	**1**	**0**	**0**	**0**	**0**	**0**	*Not*	*cc*					**0**	**0**	**0**	**0**	**0**
	x	*pma16*															
J*cc* *pma16*, **R***x***++**	**1**	**0**	**0**	**0**	**0**	**0**	*Not*	*cc*					**R***x*			**0**	**1**
	x	*pma16*															
J*cc* *pma16*, **R***x***––**	**1**	**0**	**0**	**0**	**0**	**0**	*Not*	*cc*					**R***x*			**1**	**0**
	x	*pma16*															
J*cc* *pma16*, **R***x***++R5**	**1**	**0**	**0**	**0**	**0**	**0**	*Not*	*cc*					**R***x*			**1**	**1**
	x	*pma16*															
MOV {*adrs*}, **A***n*[~] [, *next A*]	**0**	**0**	**1**	**1**	**A~**	*next A*		**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV A*n*[~], {*adrs*} [*, next A*]	**0**	**0**	**1**	**0**	**A~**	*next A*		**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV {*adrs*}, ***A***n*	**0**	**1**	**0**	**1**	**1**	**1**	**0**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV A*n*[~], *imm16* [*, next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**0**	**0**	**1**	**0**	**0**	**1**	**0**	**~A**
MOV MR, *imm16* [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**1**	**1**	**1**	**0**	**0**	**1**	**0**	**0**
MOV A*n*, **A***n*~ [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**0**	**0**	**1**	**1**	**1**	**0**	**A~**	**~A**
MOV A*n*[~], **PH** [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**0**	**1**	**1**	**1**	**0**	**0**	**A~**	**~A**
MOV SV, **A***n*[~] [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**1**	**0**	**1**	**0**	**0**	**0**	**A~**	0
MOV PH, **A***n*[~] [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**1**	**0**	**1**	**0**	**1**	**0**	**A~**	0
MOV A*n*[~], ***A***n*[~] [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**0**	**0**	**0**	**1**	**0**	**0**	**A~**	**~A**
MOV MR, **A***n*[~] [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**1**	**0**	**1**	**1**	**0**	**0**	**A~**	0
MOV {*adrs*}, **R***x*	**1**	**1**	**1**	**1**	**0**	**0**	**R***x*			{*adrs*}							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV R*x*, {*adrs*}	**1**	**1**	**1**	**1**	**0**	**1**	**R***x*			{*adrs*}							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV R*x*, *imm16*	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**1**	**0**	**R***x*			**0**	**0**
MOV R*x*, **R5**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**0**	**R***x*			**0**	**0**
MOV SV, *imm4*	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	*imm4*			
MOV SV, {*adrs*}†	**1**	**1**	**0**	**1**	**1**	**0**	**0**	**0**	**0**	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															

† Signed multiplier mode resets UM (bit 1 in status register) to 0

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOV PH, {*adrs*}	**1**	**1**	**0**	**1**	**1**	**0**	**0**	**0**	**1**	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV MR, {*adrs*}	**1**	**1**	**0**	**1**	**1**	**1**	**0**	**0**	**0**	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV AP*n*, {*adrs*}	**1**	**1**	**0**	**1**	**1**	**0**	**1**	**AP***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV STAT, {*adrs*}	**1**	**1**	**0**	**1**	**1**	**1**	**1**	**1**	**1**	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV TOS, {*adrs*}	**1**	**1**	**0**	**1**	**1**	**0**	**0**	**1**	**0**	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV {*adrs*}, **PH**	**1**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV {*adrs*}, **MR**	**1**	**1**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV {*adrs*}, **STAT**	**1**	**1**	**0**	**1**	**0**	**0**	**0**	**1**	**0**	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV {*adrs*}, **STR**	**1**	**1**	**0**	**1**	**0**	**0**	**0**	**1**	**1**	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV {*adrs*}, **DP**	**1**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV {adrs}, **SV**	**1**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV {adrs}, **AP***n*	**1**	**1**	**0**	**1**	**0**	**0**	**1**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV {*adrs*}, **TOS**	**1**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**1**	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV STR, {*adrs*}	**1**	**1**	**0**	**1**	**1**	**0**	**0**	**1**	**1**	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOV {*flagadrs*}†, **TF***n*	**1**	**0**	**0**	**1**	**1**	*flg*	*Not*	**0**	**0**	**1**	*flagadrs*						
MOV TF*n*, {*flagadrs*}†	**1**	**0**	**0**	**1**	**1**	*flg*	*Not*	**0**	**0**	**0**	*flagadrs*						
MOV TF*n*, {*cc*} [, **R***x*]	**1**	**0**	**0**	**1**	**0**	*flg*	*Not*	*cc*					**R***x*			**0**	**0**
MOV STR, *imm8*	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**1**	*imm8*							
MOV AP*n*, *imm6*	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**A***n*		**0**	**0**	**0**	*imm5*				
MOVB A*n*, {*adrs*}	**0**	**1**	**0**	**0**	**1**	**1**	**0**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOVB {*adrs*}, **A***n*	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**A***n*		*adrs*							

† *Flagadrs* is 64 locations (global or relative to R6)

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOVB {*adrs*}, **A***n*	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOVB A*n*, *imm8*	**1**	**0**	**1**	**0**	**0**	**0**	**1**	**A***n*		*imm8*							
MOVB MR, *imm8*	**1**	**0**	**1**	**0**	**1**	**1**	**1**	**A***n*		*imm8*							
MOVB R*x*, *imm8*	**1**	**0**	**1**	**1**	**1**	**0**	*k4*	*k3*	*k2*	*k7*	*k6*	*k5*	**R***x*			*k1*	*k0*
MOVBS A*n*, {*adrs}*	**0**	**1**	**0**	**0**	**1**	**1**	**1**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOVBS {*adrs}*, **A***n*	**0**	**1**	**0**	**1**	**0**	**0**	**1**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOVS A*n*[~], {*adrs*}	**0**	**0**	**1**	**0**	**A~**	**1**	**1**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOVS {*adrs*}, **A***n*[~]	**0**	**0**	**1**	**1**	**A~**	**1**	**1**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOVS {*adrs*}, ***A***n*	**0**	**1**	**0**	**1**	**1**	**1**	**1**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOVS A*n*[~], *pma16*	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**0**	**0**	**1**	**0**	**0**	**1**	**A~**	**~A**
MOVS PH, **A***n*[~]	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**1**	**0**	**1**	**0**	**1**	**0**	**A~**	**0**
MOVS SV, **A***n*[~]	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**1**	**0**	**1**	**0**	**0**	**0**	**A~**	**0**
MOVS A*n*[~], **PH**	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**0**	**1**	**1**	**1**	**0**	**0**	**A~**	**~A**
MOVS A*n*, **A***n~*	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**0**	**0**	**1**	**1**	**1**	**0**	**A~**	**~A**
MOVS MR, **A***n*[~]	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**1**	**0**	**1**	**1**	**0**	**0**	**A~**	**0**
MOVS A*n*[~], ***A***n*[~]	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**0**	**0**	**0**	**1**	**0**	**0**	**A~**	**~A**
MOVT {*adrs*}, **TF***n*	**1**	**1**	**0**	**1**	**0**	**1**	**1**	**1**	*flg*	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOVU MR, **A***n*[~] [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**1**	**0**	**1**	**1**	**1**	**0**	**A~**	**0**
MOVU MR, {*adrs*}	**1**	**1**	**0**	**1**	**1**	**1**	**0**	**0**	**1**	*adrs*							
	0	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOVAPH A*n*, **MR**, {*adrs*}	**0**	**1**	**1**	**0**	**1**	**0**	**0**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOVAPHS A*n*, **MR**, {*adrs*}	**0**	**1**	**1**	**0**	**1**	**0**	**1**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOVSPH A*n*, **MR**, {*adrs*}	**0**	**1**	**1**	**0**	**0**	**1**	**0**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MOVSPHS A*n*, **MR**, {*adrs*}	**0**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MUL A*n*[~] [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**1**	**1**	**1**	**1**	**0**	**0**	**A~**	**0**

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MUL {*adrs*}	**1**	**1**	**0**	**1**	**1**	**1**	**0**	**1**	**1**	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MULR {*adrs*}	**1**	**1**	**0**	**1**	**1**	**1**	**0**	**1**	**0**	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MULS A*n*[~]	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**1**	**1**	**1**	**1**	**0**	**0**	**A~**	**0**
MULAPL A*n*, {*adrs*}	**0**	**1**	**1**	**0**	**1**	**1**	**0**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MULAPL A*n*[~], **A***n*[~], [*next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**1**	**1**	**0**	**0**	**1**	**0**	**A~**	**~A**
MULAPLS A*n*, {*adrs*}	**0**	**1**	**1**	**0**	**1**	**1**	**1**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MULAPLS A*n*[~], **A***n*[~]	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**1**	**1**	**0**	**0**	**1**	**0**	**A~**	**~A**
MULSPL A*n*, {*adrs*}	**0**	**1**	**1**	**1**	**1**	**1**	**0**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MULSPL A*n*[~], **A***n*[~] [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**1**	**1**	**0**	**0**	**0**	**0**	**A~**	**~A**
MULSPLS A*n*, {*adrs*}	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MULSPLS A*n*[~], **A***n*[~]	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**1**	**1**	**0**	**0**	**0**	**0**	**A~**	**~A**
MULTPL A*n*, {*adrs*}	**0**	**1**	**1**	**0**	**0**	**0**	**0**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MULTPL A*n*[~], **A***n*[~] [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**1**	**1**	**0**	**1**	**1**	**0**	**A~**	**~A**
MUL TPLS A*n*, {*adrs*}	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
MULTPLS A*n*[~], **A***n*[~]	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**1**	**1**	**0**	**1**	**1**	**0**	**A~**	**~A**
NEGAC A*n*[~], **A***n*[~] [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**0**	**0**	**0**	**0**	**0**	**0**	**A~**	**~A**
NEGACS A*n*[~], **A***n*[~]	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**0**	**0**	**0**	**0**	**0**	**0**	**A~**	**~A**
NOTAC A*n*[~], **A***n*[~] [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**0**	**0**	**0**	**0**	**1**	**0**	**A~**	**~A**
NOTACS A*n*[~], **A***n*[~]	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**0**	**0**	**0**	**0**	**1**	**0**	**A~**	**~A**
NOP	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
OR A*n*, {*adrs*}	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
OR A*n*[~], **A***n*[~], *imm16* [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**1**	**0**	**0**	**0**	**0**	**1**	**A~**	**~A**
OR A*n*[~], **A***n*~, **A***n* [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**0**	**1**	**0**	**0**	**1**	**0**	**A~**	**~A**
OR TF*n*, {*flagadrs*}	**1**	**0**	**0**	**1**	**1**	*flg*	*Not*	**0**	**1**	**0**	*flagadrs*						
OR TF*n*, {*cc*} [, **R***x*]	**1**	**0**	**0**	**1**	**0**	*flg*	*Not*	*cc*					**R***x*			**0**	**1**
ORB A*n*, *imm8*	**1**	**0**	**1**	**0**	**1**	**0**	**0**	**A***n*		*imm8*							
ORS A*n*, {*adrs*}	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ORS **A***n*[~], **A***n*[~], *pma16*	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**1**	**0**	**0**	**0**	**0**	**1**	**A~**	**~A**
ORS **A***n*[~], **A***n*~, **A***n*	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**0**	**1**	**0**	**0**	**1**	**0**	**A~**	**~A**
OUT *port4*, {*adrs*}	**1**	**1**	**0**	**0**	**1**	*port4*				*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
OUT *port6*, **A***n*[~]	**1**	**1**	**1**	**0**	**1**	**1**	**0**	**A***n*		*port6*						**1**	**A~**
OUTS *port6*, **A***n*[~]	**1**	**1**	**1**	**0**	**1**	**1**	**1**	**A***n*		*port6*						**1**	**A~**
RPT {*adrs*}	**1**	**1**	**0**	**1**	**1**	**1**	**1**	**1**	**0**	*adrs*							
RPT *imm8*	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	*imm8*							
RET	**1**	**1**	**0**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**0**
RFLAG {*flagadrs*}	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	*flagadrs*						
RFM	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**0**	**1**	**0**	**0**	**0**	**0**
ROVM	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**
RTAG {*adrs*}	**1**	**1**	**0**	**1**	**0**	**1**	**1**	**0**	**1**	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
RXM	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**0**	**1**	**1**	**0**	**0**	**0**	**0**
SFLAG {*flagadrs*}	**1**	**0**	**0**	**1**	**1**	**1**	**0**	**1**	**0**	**1**	*flagadrs*						
SFM	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**
SHL **A***n*[~] [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**1**	**1**	**1**	**1**	**1**	**0**	**A~**	**0**
SHLS **A***n*[~]	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**1**	**1**	**1**	**1**	**1**	**0**	**A~**	**0**
SHLAPL **A***n*, {*adrs*}	**0**	**1**	**1**	**1**	**1**	**0**	**0**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
SHLAPL **A***n*[~], **A***n*[~] [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**1**	**1**	**1**	**0**	**1**	**0**	**A~**	**~A**
SHLAPLS **A***n*, {*adrs*}	**0**	**1**	**1**	**1**	**1**	**0**	**1**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
SHLAPLS **A***n*[~], A*n*[~]	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**1**	**1**	**1**	**0**	**1**	**0**	**A~**	**~A**
SHLSPL **A***n*, {*adrs*}	**0**	**1**	**1**	**1**	**0**	**1**	**0**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
SHLSPL **A***n*[~], **A***n*[~] [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**1**	**1**	**1**	**0**	**0**	**0**	**A~**	**~A**
SHLSPLS **A***n*, {*adrs*}	**0**	**1**	**1**	**1**	**0**	**1**	**1**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
SHLSPLS **A***n*[~], **A***n*[~]	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**1**	**1**	**1**	**0**	**0**	**0**	**A~**	**~A**
SHLTPL **A***n*, {*adrs*}	**0**	**1**	**1**	**1**	**0**	**0**	**0**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
SHLTPL **A***n*[~], **A***n*[~] [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**1**	**1**	**0**	**1**	**0**	**0**	**A~**	**~A**

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SHLTPLS **A***n*, {*adrs*}	**0**	**1**	**1**	**1**	**0**	**0**	**1**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SHLTPLS A*n*[~], **A***n*[~]	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**1**	**1**	**0**	**1**	**0**	**0**	**A~**	**~A**
SHLAC A*n*[~], **A***n*[~] [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**0**	**0**	**1**	**1**	**0**	**0**	**A~**	**~A**
SHLACS A*n*[~], **A***n*[~]	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**0**	**0**	**1**	**1**	**0**	**0**	**A~**	**~A**
SHRAC A*n*[~], **A***n*[~] [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**0**	**1**	**0**	**1**	**1**	**0**	**A~**	**~A**
SHRACS A*n*[~], **A***n*[~]	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**0**	**1**	**0**	**1**	**1**	**0**	**A~**	**~A**
STAG {*adrs*}	**1**	**1**	**0**	**1**	**0**	**1**	**1**	**0**	**0**	*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
SOVM	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**1**	**0**	**1**	**0**	**0**	**0**	**0**
SUB A*n*[~], **A***n*, {*adrs*} [, *next A*]	**0**	**0**	**0**	**1**	**~A**	*next A*		**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
SUB A*n*[~], **A***n*[~], *imm16* [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**0**	**1**	**0**	**0**	**0**	**1**	**A~**	**~A**
SUB A*n*[~], **A***n*[~], **PH** [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**0**	**1**	**1**	**0**	**0**	**0**	**A~**	**~A**
SUB A*n*[~], **A***n*, **A***n*~ [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**0**	**0**	**1**	**0**	**0**	**0**	**0**	**~A**
SUB A*n*[~], **A***n*~, **A***n* [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**0**	**0**	**1**	**0**	**0**	**0**	**1**	**~A**
SUB R*x*, *imm16*	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**1**	**R***x*			**0**	**0**
SUB R*x*, **R5**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**1**	**0**	**1**	**R***x*			**0**	**0**
SUBB A*n*, *imm8*	**1**	**0**	**1**	**0**	**0**	**1**	**0**	**A***n*		*imm8*							
SUBB R*x*, *imm8*	**1**	**0**	**1**	**1**	**0**	**1**	*k4*	*k3*	*k2*	*k7*	*k6*	*k5*	**R***x*			*k1*	*k0*
SUBS A*n*[~], **A***n*, {*adrs*}	**0**	**0**	**0**	**1**	**~A**	**1**	**1**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
SUBS A*n*[~], **A***n*[~], *pma16*	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**0**	**1**	**0**	**0**	**0**	**1**	**A~**	**~A**
SUBS A*n*[~], **A***n*, **A***n*~	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**0**	**0**	**1**	**0**	**0**	**0**	**0**	**~A**
SUBS A*n*[~], **A***n*~, **A***n*	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**0**	**0**	**1**	**0**	**0**	**0**	**1**	**~A**
SUBS A*n*[~], **A***n*[~], **PH**	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**0**	**1**	**1**	**0**	**0**	**0**	**A~**	**~A**
SXM	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**0**
VCALL *vector8*	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	*vector8*							
XOR A*n*, {*adrs*}	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
XOR A*n*[~], **A***n*[~], *imm16* [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**1**	**1**	**0**	**0**	**0**	**1**	**A~**	**~A**
XOR A*n*[~], **A***n*~, **A***n* [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**0**	**1**	**0**	**0**	**0**	**0**	**A~**	**~A**
XOR TF*n*, {*flagadrs*}	**1**	**0**	**0**	**1**	**1**	*flg*	*Not*	**1**	**1**	**0**	*flagadrs*						
XOR TF*n*, {*cc*} [, **R***x*]	**1**	**0**	**0**	**1**	**0**	*flg*	*Not*	*cc*					**R***x*			**1**	**1**
XORB A*n*, *imm8*	**1**	**0**	**1**	**0**	**1**	**1**	**0**	**A***n*		*imm8*							
XORS A*n*, {*adrs*}	**0**	**1**	**0**	**0**	**1**	**0**	**1**	**A***n*		*adrs*							
	x	*dma16* (for direct) or *offset16* (long relative) [see section 4.13]															
XORS A*n*[~], **A***n*[~], *pma16*	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**1**	**1**	**0**	**0**	**0**	**1**	**A~**	**~A**
XORS A*n*[~], **A***n*~, **A***n*	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**0**	**1**	**0**	**0**	**0**	**0**	**A~**	**~A**

Instructions	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ZAC A*n*[~] [, *next A*]	**1**	**1**	**1**	**0**	**0**	*next A*		**A***n*		**0**	**0**	**0**	**1**	**1**	**0**	**0**	**~A**
ZACS A*n*[~]	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**A***n*		**0**	**0**	**0**	**1**	**1**	**0**	**0**	**~A**

cc					**cc** names		**Description** **True condition (*Not* true condition)**
					cc name	**Not cc name**	
0	0	0	0	0	**Z**	**NZ**	Conditional on **ZF**=1 (*Not* condition **ZF**=0)
0	0	0	0	1	**S**	**NS**	Conditional on **SF**=1 (*Not* condition **SF**=0)
0	0	0	1	0	**C**	**NC**	Conditional on **CF**=1 (*Not* condition **CF**=0)
0	0	0	1	1	**B**	**NB**	Conditional on **ZF**=0 and **CF**=0 (*Not* condition **ZF**≠0 or **CF**≠0)
0	0	1	0	0	**A**	**NA**	Conditional on **ZF**=0 and **CF**=1 (*Not* condition **ZF**≠0 or **CF**≠1)
0	0	1	0	1	**G**	**NG**	Conditional on **SF**=0 and **ZF**=0 (*Not* condition **SF**≠0 or **ZF**≠0)
0	0	1	1	0	**E**	**NE**	Conditional if **ZF**=1 and **OF**=0 (*Not* condition **ZF**≠1 or **OF**≠0)
0	0	1	1	1	**O**	**NO**	Conditional if **OF**=1 (*Not* condition **OF**=0)
0	1	0	0	0	**RC**	**RNC**	Conditional on **RCF**=1 (*Not* condition **RCF**=0)
0	1	0	0	1	**RA**	**RNA**	Conditional on **RZF**=0 and **RCF**=1 (*Not* condition **RZF**≠0 or **RCF**≠1)
0	1	0	1	0	**RE**	**RNE**	Conditional on **RZF**=1 (*Not* condition **RZF**=0)
0	1	0	1	1	**RZP**	**RNZP**	Conditional on value of **R***x*=0 Not available on Calls. (*Not* condition **R***x*≠0)
0	1	1	0	0	**RLZP**	**RNLZP**	Conditional on MSB of **R**x=1. Not available on Calls. (*Not* condition MSB of **R***x*=0)
0	1	1	0	1	**L**	**NL**	Conditional on **ZF**=0 and **SF**=1 (*Not* condition **ZF**≠0 or **SF**≠1)
0	1	1	1	0			reserved
0	1	1	1	1			reserved
1	0	0	0	0	**TF1**	**NTF1**	Conditional on **TF1**=1 (*Not* condition **TF1**=0)
1	0	0	0	1	**TF2**	**NTF2**	Conditional on **TF2**=1 (*Not* condition **TF2**=0)
1	0	0	1	0	**TAG**	**NTAG**	Conditional on **TAG**=1 (*Not* condition **TAG**=0)
1	0	0	1	1	**IN1**	**NIN1**	Conditional on **IN1**=1 status. (*Not* condition **IN1**=0)
1	0	1	0	0	**IN2**	**NIN2**	Conditional on **IN2**=1 status. (*Not* condition **IN2**=0)
1	0	1	0	1			Unconditional
1	0	1	1	0			reserved
1	0	1	1	1			reserved
1	1	0	0	0	**XZ**	**XNZ**	Conditional on **XZF**=1 (*Not* condition **XZF**=0)
1	1	0	0	1	**XS**	**XNS**	Conditional on **XSF**=1 (*Not* condition **XSF**=0)
1	1	0	1	0	**XG**	**XNG**	Conditional on **XSF**=0 and **XZF**=0 (*Not* condition **XSF**≠0 or **XZF**≠0)
1	1	0	1	1			reserved
1	1	1	0	0			reserved
1	1	1	0	1			reserved
1	1	1	1	0			reserved
1	1	1	1	1			reserved

4.16 Instruction Set Summary

Use the legend in Section 4.13 and the following table to obtain a summary of each instruction and its format. For detail about the instruction refer to the detail description of the instruction.

name	*dest [, src] [, src1] [,mod]*	**Clock**, *clk*	**Words**, *w*	**With RPT**, *clk*	**Class**
ADD	**A***n*[~], **A***n*, {*adrs*} [, *next A*]	Table 4–46	Table 4–46	Table 4–46	1a
ADD	**A***n*[~], **A***n*[~], *imm16* [, *next A*]	2	2	N/R	2b
ADD	**A***n*[~], **A***n*[~], **PH** [, *next A*]	1	1	n_R+3	3
ADD	**A***n*[~], **A***n*~, **A***n* [, *next A*]	1	1	n_R+3	3
ADD	**R***x*, *imm16*	2	2	N/R	4c
ADD	**R***x*, R5	1	1	n_R+3	4d
ADD	**A**P*n*, *imm5*	1	1	N/R	9c
ADDB	**A***n*, *imm8*	1	1	N/R	2a
ADDB	**R***x*, *imm8*	1	1	N/R	4b
ADDS	**A***n*[~], **A***n*, {*adrs*}	Table 4–46	Table 4–46	Table 4–46	1a
ADDS	**A***n*[~], **A***n*[~], *pma16*	n_S+4	2	N/R	2b
ADDS	**A***n*[~], **A***n*~, **A***n*	n_S+3	1	n_R+3	3
ADDS	**A***n*[~], **A***n*[~], **PH**	n_S+3	1	n_R+3	3
AND	**A***n*, {*adrs*}	Table 4–46		Table 4–46	1b
AND	**A***n*[~], **A***n*[~], *imm16* [, *next A*]	2	2	N/R	2b
AND	**A***n*[~], **A***n*~, **A***n* [, *next A*]	1	1	n_R+3	3
AND	**TF***n*, {*flagadrs*}	1	1	n_R+3	8a
AND	**TF***n*, {*cc*} [, **R***x*]	1	1	n_R+3	8b
ANDB	**A***n*, *imm8*	1	1	N/R	2a
ANDS	**A***n*, {*adrs*}	Table 4–46		Table 4–46	1b
ANDS	**A***n*[~], **A***n*[~], *pma16*	n_S+4	1	N/R	2b
ANDS	**A***n*[~], **A***n*~, **A***n*	n_S+3	1	n_R+3	3
BEGLOOP		1	1	N/R	9d
CALL	*pma16*	2	2	N/R	7c
CALL	***A***n*	2	2	N/R	7c
C*cc*	*pma16*	2	2	N/R	7c
CMP	**A***n*, {*adrs*}	Table 4–46		Table 4–46	1b

name	*dest [, src] [, src1] [,mod]*	**Clock**, *clk*	**Words**, *w*	**With RPT**, *clk*	**Class**
CMP	**R***x*, *imm16*	2	2	N/R	2b
CMP	**A***n*[~], **A***n*[~] [, *next A*]	1	1	N/R	3
CMP	**A***n*[~], *imm16* [, *next A*]	2	2	N/R	4c
CMP	**R***x*, R5	1	1	N/R	4d
CMPB	**A***n*, *imm8*	1	1	N/R	2a
CMPB	**R***x*, *imm8*	1	1	N/R	4b
CMPS	**A***n*, {*adrs*}	Table 4–46		Table 4–46	1b
CMPS	**A***n*[~], *pma16*	n_S+4	2	N/R	2b
CMPS **CMPS**	**A***n*, **A***n*~ **A***n*~, **A***n*	n_S+3	1	n_R+3	3
COR	**A***n*, ***R***x*	3	1	3(n_R+2)	9a
CORK	**A***n*, ***R***x*	3	1	3(n_R+2)	9a
ENDLOOP	*n*	1	1	N/R	9d
EXTSGN	**A***n*[~] [, *next A*]	1	1	n_R+3	3
EXTSGNS	**A***n*[~]	n_S+3	1	n_R+3	3
FIR	**A***n*, ***R***x*	2	1	2(n_R+2)	9a
FIRK	**A***n*, ***R***x*	2	1	2(n_R+2)	9a
IDLE		1	1	N/R	9d
IN	{*adrs*}, *port4*	Table 4–46		Table 4–46	6a
IN	**A***n*[~], *port6*	1	1	N/R	6b
INS	**A***n*[~], *port6*	n_S+4	2	n_R+4	6b
INTD		1	1	N/R	9d
INTE		1	1	N/R	9d
IRET		2	1	N/R	5
JMP	*pma16*	2	2	N/R	7b
JMP	*pma16*, **R***x***++**	2	2	N/R	7b
JMP	*pma16*, **R***x***––**	2	2	N/R	7b
JMP	*pma16*, **R***x***++R5**	2	2	N/R	7b
JMP	***A***n*	2	1	N/R	7b
J*cc*	*pma16* [, *Rmod]*	2	2	N/R	7b

name	*dest [, src] [, src1] [,mod]*	**Clock**, *clk*	**Words**, *w*	**With RPT**, *clk*	**Class**
MOV	{*adrs*}, **A***n*[~] [, *next A*]	Table 4–46		Table 4–46	1a
MOV	**A***n*[~], {*adrs*} [, *next A*]	Table 4–46		Table 4–46	1a
MOV	{*adrs*}, ***A***n*	Table 4–46		Table 4–46	1b
MOV	**A***n*[~], *imm16* [, *next A*]	2	2	N/R	2b
MOV	**MR**, *imm16* [, *next A*]	2	2	N/R	2b
MOV	**A***n*, **A***n*~ [, *next A*]	1	1	n_R+3	3
MOV	**A***n*[~], **PH** [, *next A*]	1	1	n_R+3	3
MOV	**SV, A***n*[~] [, *next A*]	1	1	n_R+3	3
MOV	**PH, A***n*[~] [, *next A*]	1	1	n_R+3	3
MOV	**A***n*[~], ***A***n*[~] [, *next A*]	1	1	n_R+3	3
MOV	**MR, A***n*[~] [, *next A*]	1	1	n_R+3	3
MOV	{*adrs*}, **R***x*	Table 4–46		Table 4–46	4a
MOV	**R***x*, {*adrs*}	Table 4–46		Table 4–46	4a
MOV	**R***x*, *imm16*	2	2	N/R	4c
MOV	**R***x*, **R5**	1	1	n_R+3	4d
MOV	**SV**, *imm4*	1	1	N/R	5
MOV	**SV**, {*adrs*}†	1	1	n_R+3	5
MOV	**PH**, {*adrs*}	Table 4–46		Table 4–46	5
MOV	**MR**, {*adrs*}	Table 4–46		Table 4–46	5
MOV	**AP***n*, {*adrs*}	Table 4–46		Table 4–46	5
MOV	**STAT**, {*adrs*}	Table 4–46		Table 4–46	5
MOV	**TOS**, {*adrs*}	Table 4–46		Table 4–46	5
MOV	{*adrs*}, **PH**	Table 4–46		Table 4–46	5
MOV	{*adrs*}, **MR**	Table 4–46		Table 4–46	5
MOV	{*adrs*}, **STAT**	Table 4–46		Table 4–46	5
MOV	{*adrs*}, **STR**	Table 4–46		Table 4–46	5
MOV	{*adrs*}, **DP**	Table 4–46		Table 4–46	5

† Signed multiplier mode resets UM (bit 1 in status register) to 0

name	*dest [, src] [, src1] [,mod]*	**Clock**, *clk*	**Words**, *w*	**With RPT**, *clk*	**Class**
MOV	{*adrs*}, **SV**	Table 4–46		Table 4–46	5
MOV	{*adrs*}, **AP***n*	Table 4–46		Table 4–46	5
MOV	{*adrs*}, **TOS**	Table 4–46		Table 4–46	5
MOV	**STR**, {*adrs*}	Table 4–46		Table 4–46	5
MOV	{*flagadrs*}†, **TF***n*	1	1	n_R+3	8a
MOV	**TF***n*, {*flagadrs*}†	1	1	n_R+3	8a
MOV	**TF***n*, {*cc*} [, **R***x*]	1	1	N/R	8b
MOV	**STR**, *imm8*	1	1	N/R	9b
MOV	**AP***n*, *imm5*	1	1	N/R	9c
MOVB	**A***n*, {*adrs*}†	Table 4–46		Table 4–46	1b
MOVB	{*adrs*}†, **A***n*	Table 4–46		Table 4–46	1b
MOVB	**A***n*, *imm8*	1	1	N/R	2a
MOVB	**MR**, *imm8*	1	1	N/R	2a
MOVB	**R***x*, *imm8*	1	1	N/R	4b
MOVBS	**A***n*, $\{adrs\}_{\dagger}$	Table 4–46		Table 4–46	1b
MOVBS	$\{adrs\}_8$, **A***n*	Table 4–46		Table 4–46	1b
MOVS	**A***n*[~], {*adrs*}	Table 4–46		Table 4–46	1a
MOVS	{*adrs*}, **A***n*[~]	Table 4–46		Table 4–46	1a
MOVS	{*adrs*}, ***A***n*	Table 4–46		Table 4–46	1b
MOVS	**A***n*[~], *pma16*	n_S+4	2	N/R	2b
MOVS	**PH, A***n*[~]	n_S+3	1	n_R+3	3
MOVS	**SV, A***n*[~]	n_S+3	1	n_R+3	3
MOVS	**A***n*[~], **PH**	n_S+3	1	n_R+3	3
MOVS	**A***n*, **A***n*~	n_S+3	1	n_R+3	3
MOVS	**MR, A***n*[~]	n_S+3	1	n_R+3	3
MOVS	**A***n*[~], ***A***n*[~]	n_S+3	1	n_R+3	3
MOVT	{*adrs*}, **TF***n*	Table 4–46		Table 4–46	5
MOVU	**MR, A***n*[~] [, *next A*]	1	1	n_R+3	3

† *Flagadrs* is 64 locations (global or relative to R6)

name	*dest [, src] [, src1] [,mod]*	**Clock**, *clk*	**Words**, *w*	**With RPT**, *clk*	**Class**
MOVU	**MR**, {*adrs*}	Table 4–46		Table 4–46	5
MOVAPH	**A***n*, **MR**, {*adrs*}	Table 4–46		Table 4–46	1b
MOVAPHS	**A***n*, **MR**, {*adrs*}	Table 4–46		Table 4–46	1b
MOVSPH	**A***n*, **MR**, {*adrs*}	Table 4–46		Table 4–46	1b
MOVSPHS	**A***n*, **MR**, {*adrs*}	Table 4–46		Table 4–46	1b
MUL	**A***n*[~] [, *next A*]	1	1	n_R+3	3
MUL	{*adrs*}	Table 4–46		Table 4–46	5
MULR	{*adrs*}	Table 4–46		Table 4–46	5
MULS	**A***n*[~]	n_S+3	1	n_R+3	3
MULAPL	**A***n*, {*adrs*}	Table 4–46		Table 4–46	1b
MULAPL	**A***n*[~], **A***n*[~] [, *next A*]	1	1	n_R+3	3
MULAPLS	**A***n*, {*adrs*}	Table 4–46		Table 4–46	1b
MULAPLS	**A***n*[~]**, A***n*[~]	n_S+3	1	n_R+3	3
MULSPL	**A***n*, {*adrs*}	Table 4–46		Table 4–46	1b
MULSPL	**A***n*[~], **A***n*[~] [, *next A*]	1	1	n_R+3	3
MULSPLS	**A***n*, {*adrs*}	Table 4–46		Table 4–46	1b
MULSPLS	**A***n*[~]**,** A*n*[~]	n_S+3	1	n_R+3	3
MULTPL	**A***n*, {*adrs*}	Table 4–46		Table 4–46	1b
MULTPL	**A***n*[~], **A***n*[~] [, *next A*]	1	1	n_R+3	3
MULTPLS	**A***n*, {*adrs*}	Table 4–46		Table 4–46	1b
MULTPLS	**A***n*[~] **, A***n*[~]	n_S+3	1	n_R+3	3
NEGAC	**A***n*[~], **A***n*[~] [, *next A*]	1	1	n_R+3	3
NEGACS	**A***n*[~] **, A***n*[~]	n_S+3	1	n_R+3	3
NOTAC	**A***n*[~], **A***n*[~] [, *next A*]	1	1	n_R+3	3
NOTACS	**A***n*[~] **, A***n*[~]	n_S+3	1	n_R+3	3
NOP		1	1	n_R+3	9d
OR	A*n*, {*adrs*}	Table 4–46		Table 4–46	1b
OR	**A***n*[~], **A***n*[~], *imm16* [, *next A*]	2	2	N/R	2b
OR	**A***n*[~], **A***n*~, **A***n* [, *next A*]	1	1	n_R+3	3

name	*dest [, src] [, src1] [,mod]*	**Clock**, *clk*	**Words**, *w*	**With RPT**, *clk*	**Class**
OR	**TF***n*, {*flagadrs*}	1	1	n_R+3	8a
OR	**TF***n*, {*cc*} [, **R***x*]	1	1	N/R	8b
ORB	**A***n*, *imm8*	1	1	N/R	2a
ORS	**A***n*, {*adrs*}	Table 4–46		Table 4–46	1b
ORS	**A***n*[~], **A***n*[~], *pma16*	n_S+4	2	N/R	2b
ORS	**A***n*[~], **A***n*~, **A***n*	n_S+3	1	n_R+3	3
OUT	*port4*, {adrs}	Table 4–46		n_R+3	6a
OUTS	*port6*, **A**n[~]	n_S+3	1	n_R+3	6b
RPT	{*adrs*}$_8$	Table 4–46		N/R	5
RPT	*imm8*	1	1	N/R	9b
RET		1	1	N/R	5
RFLAG	{*flagadrs*}	1	1	n_R+3	8a
RFM		1	1	n_R+3	9d
ROVM		1	1	N/R	9d
RTAG	{*adrs*}	Table 4–46		Table 4–46	5
RXM		1	1	N/R	9d
SFLAG	{*flagadrs*}	1	1	n_R+3	8a
SFM		1	1	N/R	9d
SHL	**A***n*[~] [, *next A*]	1	1	n_R+3	3
SHLS	**A***n*[~]	n_S+3	1	n_R+3	3
SHLAPL	**A***n*, {*adrs*}	Table 4–46		Table 4–46	1b
SHLAPL	**A***n*[~], **A***n*[~] [, *next A*]	1	1	n_R+3	3
SHLAPLS	**A***n*, {*adrs*}	Table 4–46		Table 4–46	1b
SHLAPLS	**A***n*[~] **, A***n*[~]	n_S+3	1	n_R+3	3
SHLSPL	**A***n*, {*adrs*}	Table 4–46		Table 4–46	1b
SHLSPL	**A***n*[~], **A***n*[~] [, *next A*]	1	1	n_R+3	3
SHLSPLS	**A***n*, {*adrs*}	Table 4–46		Table 4–46	1b
SHLSPLS	**A***n*[~] , **A***n*[~]	n_S+3	1	n_R+3	3
SHLTPL	**A***n*, {*adrs*}	Table 4–46		Table 4–46	1b
SHLTPL	**A***n*[~], **A***n*[~] [, *next A*]	1	1	n_R+3	3

name	*dest [, src] [, src1] [,mod]*	**Clock**, *clk*	**Words**, *w*	**With RPT**, *clk*	**Class**
SHLTPLS	**A***n*, {*adrs*}	Table 4–46		Table 4–46	1b
SHLTPLS	**A***n*[~], **A***n*[~]	n_S+3	1	n_R+3	3
SHLAC	**A***n*[~], **A***n*[~] [, *next A*]	1	1	n_R+3	3
SHLACS	**A***n*[~], **A***n*[~]	n_S+3	1	n_R+3	3
SHRAC	**A***n*[~], **A***n*[~] [, *next A*]	1	1	n_R+3	3
SHRACS	**A***n*[~], **A***n*[~]	n_S+3	1	n_R+3	3
STAG	{*adrs*}	Table 4–46		Table 4–46	5
SOVM		1	1	N/R	9d
SUB	**A***n*[~], **A***n*, {*adrs*} [, *next A*]	Table 4–46		Table 4–46	1a
SUB	**A***n*[~], **A***n*[~], *imm16* [, *next A*]	2	2	N/R	2b
SUB	**A***n*[~], **A***n*[~], **PH** [, *next A*]	1	1	n_R+3	3
SUB	**A***n*[~], **A***n*, **A***n*~ [, *next A*]	1	1	n_R+3	3
SUB	**A***n*[~], **A***n*~, **A***n* [, *next A*]	1	1	n_R+3	3
SUB	**R***x*, *imm16*	2	2	N/R	4c
SUB	**R***x*, **R5**	1	1	n_R+3	4d
SUBB	**A***n*, *imm8*	1	1	N/R	2a
SUBB	**R***x*, *imm8*	1	1	N/R	4b
SUBS	**A***n*[~], **A***n*, {*adrs*}	Table 4–46		Table 4–46	1a
SUBS	**A***n*[~], **A***n*[~], *pma16*	2	2	N/R	2b
SUBS	**A***n*[~], **A***n*, **A***n*~	1	1	n_R+3	3
SUBS	**A***n*[~], **A***n*~, **A***n*	1	1	n_R+3	3
SUBS	**A***n*[~], **A***n*[~], **PH**	1	1	n_R+3	3
SXM		1	1	N/R	9d
VCALL	*vector8*	2	1	N/R	7a
XOR	**A***n*, {*adrs*}	Table 4–46		Table 4–46	1b
XOR	**A***n*[~], **A***n*[~], *imm16* [, *next A*]	2	2	N/R	2b
XOR	**A***n*[~], **A***n*~, **A***n* [, *next A*]	1	1	n_R+3	3
XOR	**TF***n*, {*flagadrs*}	1	1	n_R+3	8a
XOR	**TF***n*, {*cc*} [, **R***x*]	1	1	n_R+3	8b
XORB	**A***n*, *imm8*	1	1	N/R	2a

name	*dest [, src] [, src1] [,mod]*	**Clock**, *clk*	**Words**, *w*	**With RPT**, *clk*	**Class**
XORS	**A***n*, {*adrs*}	Table 4–46		Table 4–46	1b
XORS	**A***n*[~], **A***n*[~], *pma16*	n_S+4	2	N/R	2b
XORS	**A***n*[~], **A***n*~, **A***n*	n_S+3	1	n_R+3	3
ZAC	**A***n*[~] [, *next A*]	1	1	n_R+3	3
ZACS	**A***n*[~]	n_S+3	1	n_R+3	3

cc names		**Description** **True Condition (*Not* true condition)**
cc name	*Not* **cc** name	
Z	**NZ**	Conditional on **ZF**=1 (*Not* condition **ZF**=0)
S	**NS**	Conditional on **SF**=1 (*Not* condition **SF**=0)
C	**NC**	Conditional on **CF**=1 (*Not* condition **CF**=0)
B	**NB**	Conditional on **ZF**=0 and **CF**=0 (*Not* condition **ZF**≠0 or **CF**≠0)
A	**NA**	Conditional on **ZF**=0 and **CF**=1 (*Not* condition **ZF**≠0 or **CF**≠1)
G	**NG**	Conditional on **SF**=0 and **ZF**=0 (*Not* condition **SF**≠0 or **ZF**≠0)
E	**NE**	Conditional if **ZF**=1 and **OF**=0 (*Not* condition **ZF**≠1 or **OF**≠0)
O	**NO**	Conditional if **OF**=1 (*Not* condition **OF**=0)
RC	**RNC**	Conditional on **RCF**=1 (*Not* condition **RCF**=0)
RA	**RNA**	Conditional on **RZF**=0 and **RCF**=1 (*Not* condition **RZF**≠0 or **RCF**≠1)
RE	**RNE**	Conditional on **RZF**=1 (*Not* condition **RZF**=0)
RZP	**RNZP**	Conditional on value of **R***x*=0 (*Not* condition **R***x*≠0) [Not available on Calls]
RLZP	**RNLZP**	Conditional on MSB of **R**x=1. (*Not* condition MSB of **R***x*=0) [Not available on Calls]
L	**NL**	Conditional on **ZF**=0 and **SF**=1 (*Not* condition **ZF**≠0 or **SF**≠1)
TF1	**NTF1**	Conditional on **TF1**=1 (*Not* condition **TF1**=0)
TF2	**NTF2**	Conditional on **TF2**=1 (*Not* condition **TF2**=0)
TAG	**NTAG**	Conditional on **TAG**=1 (*Not* condition **TAG**=0)
IN1	**NIN1**	Conditional on **IN1**=1 status. (*Not* condition **IN1**=0)
IN2	**NIN2**	Conditional on **IN2**=1 status. (*Not* condition **IN2**=0)
XZ	**XNZ**	Conditional on **XZF**=1 (*Not* condition **XZF**=0)
XS	**XNS**	Conditional on **XSF**=1 (*Not* condition **XSF**=0)
XG	**XNG**	Conditional on **XSF**=0 and **XZF**=0 (*Not* condition **XSF**≠0 or **XZF**≠0)

MSP50C614 (MSP50P614) IO Port Description

Address	Bits	Name	R/W	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	After RESET
0x00	8	Port A Data (bidirectional)	R/W									A7	A6	A5	A4	A3	A2	A1	A0	external input states
				bit A_X = 0 ⇒ PA_X low								bit A_X = 1 ⇒ PA_X high								
0x04	8	Port A Control	R/W									C	C	C	C	C	C	C	C	0x00
				bit C = 0 ⇒ PA_X as input								bit C = 1 ⇒ PA*x* as output								
0x08	8	Port B Data (bidirectional)	R/W									B7	B6	B5	B4	B3	B2	B1	B0	external input states
				bit B_X = 0 ⇒ PB_X low								bit B_X = 1 ⇒ PB_X high								
0x0C	8	Port B Control	R/W									C	C	C	C	C	C	C	C	0x00
				bit C = 0 ⇒ PB_X as input								bit C = 1 ⇒ PB_X as output								
0x10	8	Port C Data (bidirectional)	R/W									C7	C6	C5	C4	C3	C2	C1	C0	external input states
				bit C_X = 0 ⇒ PC_X low								bit C_X = 1 ⇒ PC_X high								
0x14	8	Port C Control	R/W									C	C	C	C	C	C	C	C	0x00
				bit C = 0 ⇒ PC_X as input								bit C = 1 ⇒ PC_X as output								
0x18	8	Port D Data multifunction port (bidirectional)	R/W									D7	D6	D5†	D4†	D3	D2	D1	D0	external input states
				↓ falling edge			↑ rising edge			bit D_X = 0 ⇒ PD_X low				bit D_X = 1 ⇒ PD_X high						
				†PD_4 = inverting and PD_5 = positive comparator inputs if CE=1 in IO 0x38																
				PD_4↑ triggers INT6				PD_5↓ triggers INT7				PD_2↑ triggers INT3				PD_3↓ triggers INT4				
0x1C	8	Port D Control multifunction control	R/W									C	C	C‡	C‡	C‡	C‡	C	C	0x00
				‡C=0 for interrupts (IO 0x18)					bit C = 0 ⇒ PD_X as input					bit C = 1 ⇒ PD_X as output						
0x20	8	Port E Data (bidirectional)	R/W									E7	E6	E5	E4	E3	E2	E1	E0	external input states
				bit E_X = 0 ⇒ PE_X low								bit E_X = 1 ⇒ PE_X high								
0x24	8	Port E Control	R/W									C	C	C	C	C	C	C	C	0x00
				bit C = 0 ⇒ PD_X as input								bit C = 1 ⇒ PD_X as output								
0x28	8	Port F Data (input only)	R									F7	F6	F5	F4	F3	F2	F1	F0	external input states
				F*x* ↓ triggers INT5				bit F_X = 0 ⇒ input PF_X low						bit F_X = 1 ⇒ input PF_X high						
0x2C	16	Port G Data (output only)	R/W	G15	G14	G13	G12	G11	G10	G9	G8	G7	G6	G5	G4	G3	G2	G1	G0	0x00 all 0 outputs
				bit G*x* = 0 ⇒ PG*x* low (output only)								bit G*x* = 1 ⇒ PG*x* high (output only)								
0x2F	8	RTOTRIM ‡MSP50C614 only	R												T_4	T_3	T_2	T_1	T_0	unaffected
				T_4–T_0 = Resistor trim bits										V = 1 ⇒ T_4–T_0 are valid						
0x30	16	DAC Data	R/W	S	O	O	D	D	D	D	D	D	D	D	D	D	–	–	–§	0x0000
				S	O	O	D	D	D	D	D	D	D	D	D	–	–	–	–¶	
				S	O	O	D	D	D	D	D	D	D	D	–	–	–	–	–#	
				S = sign bit			O = overflow bit					D = data bit				– = dont care				
				§ 10 bit DAC			¶ 9 bit DAC					# 8 bit DAC				see P1,P0 in IO 0x34				

MSP50C614 (MSP50P614) IO Port Description

Address	Bits	Name	R/W	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	After RESET
0x34	4	DAC Control	R/W													DM	E	P1	P0	0x0
				DM	Drive Mode			E		Function		P1	P0	DAC bits						
				0 1	3x Style DAC 5x Style DAC			0 1		Disable DAC Enable DAC		0 1	0 1 0	8 bit 9 bits 10 bits						
0x38	16	Interrupt General Control	R/W	CE	AR	PD	EP	E2	E1	S2	S1	D5	D4	PF	D3	D2	T2	T1	DA	0x0000
				EP AR CE	F port Pullup Arm bit Comparator			Timer Function				Interrupt enable bits: 1=enable,0=disable								
				0=disable 1=enable				S1 S2	Timer1 source Timer2 source			DA T1 T2 D2 D3 PF D4 D5	DAC Timer interrupt Timer 1 interrupt Timer 2 interrupt PD2 rising edge interrupt PD3 falling edge interrupt F port falling edge interrupt PD4 rising edge interrupt PD5 falling edge interrupt							
				PD 0 1	PDM clock ½ MC MC			0 = ½ MC 1 = MC												
								E1 E2	Timer1 enable Timer2 enable											
								0 = disable 1 = enable												
0x39	8	Interrupt Flag Register	R/W									D5	D4	PF	D3	D2	T2	T1	DA	left unchanged
				D5 D4 D3 PF	PD5 falling edge interrupt flag PD4 rising edge interrupt flag PD3 falling edge interrupt flag F port falling edge interrupt flag							DA T1 T2 D2	DAC Timer interrupt flag Timer 1 interrupt flag Timer 2 interrupt flag PD2 rising edge interrupt flag							
0x3A	16	Timer 1 period	R/W		T	I	M	E	R		1		P	E	R	I	O	D		0x0000
0x3B	16	Timer 1 preset	R/W		T	I	M	E	R		1		P	R	E	S	E	T		0x0000
0x3D	16	Clock Speed Control	W	T4	T3	T2	T1	T0	I	C	R	M7	M6	M5	M4	M3	M2	M1	M0	0x0000
				Resistor Trim bits					I C R	Idle bit CRO RTO		PLLM bits MC = (PLLM value+1) × 131.07 kHz CPU clock = (PLLM value+1) × 65.536 kHz								
				0 = disable 1 = enable																
0x3E	16	Timer 2 period	R/W		T	I	M	E	R		2		P	E	R	I	O	D		0x0000
0x3F	16	Timer 2 preset	R/W		T	I	M	E	R		2		P	R	E	S	E	T		0x0000

Interrupt	Vector	Source	Trigger Event	Priority	Comment
INT0	0x7FF0	DAC Timer	timer underflow	highest	used to synch. speech data
INT1	0x7FF1	TIMER1	timer underflow	2nd	
INT2	0x7FF2	TIMER2	timer underflow	3rd	
INT3	0x7FF3	port **PD2**	rising edge	4th	port **PD2** goes HIGH
INT4	0x7FF4	port **PD3**	falling edge	5th	port **PD3** goes LOW
INT5	0x7FF5	all port **F**	any falling edge	6th	F port goes from all–HIGH to LOW
INT6†	0x7FF6	port **PD4**	rising edge	7th	port **PD4** goes HIGH
INT7†	0x7FF7	port **PD5**	falling edge	lowest	port **PD5** goes LOW
RESET	0x7FFF	hardware RESET	active low pulse	nonmaskable	Some internal I/O register

† **INT6** and **INT7** may be associated instead with the Comparator function, if the Comparator Enable bit has been set. Refer to section 3.3 for details

8 kHz Nominal Synthesis Rate (32.768 kHz oscillator reference)									
DAC Precision	**IntGenCtrl PDMCD Bit**	**Over-Sampling Factor**	**ClkSpdCtrl PLLM Register Value**	**Master Clock Rate (Hz)**	**PDM Rate (Hz)**	**CPU Clock Rate (Hz)**	**Output Sampling Rate (Hz)**	**Number of Instructs btwn DAC Interrupts**	**Number of Instructs btwn 8 kHz Interrupts**
8 bits	1	1x	0x 0F	2.10 M	2.10 M	1.05 M	8.19 k	128	128
		2x	0x 1E	4.06 M	4.06 M	2.03 M	15.87 k	128	256
		4x	0x 3E	8.26 M	8.26 M	4.13 M	32.26 k	128	512
		8x	0x 7C	16.38 M	16.38 M	8.19 M	64.00 k	128	1024
	0	1x	0x 1E	4.06 M	2.03 M	2.03 M	7.94 k	256	256
		2x	0x 3E	8.26 M	4.13 M	4.13 M	16.13 k	256	512
		4x	0x 7C	16.38 M	8.19 M	8.19 M	32.00 k	256	1024
9 bits	1	1x	0x 1E	4.06 M	4.06 M	2.03 M	7.94 k	256	256
		2x	0x 3E	8.26 M	8.26 M	4.13 M	16.13 k	256	512
		4x	0x 7C	16.38 M	16.38 M	8.19 M	32.00 k	256	1024
	0	1x	0x 3E	8.26 M	4.13 M	4.13 M	8.06 k	512	512
		2x	0x 7C	16.38 M	8.19 M	8.19 M	16.00 k	512	1024
10 bits	1	1x	0x 3E	8.26 M	8.26 M	4.13 M	8.06 k	512	512
		2x	0x 7C	16.38 M	16.38 M	8.19 M	16.00 k	512	1024
	0	1x	0x 7C	16.38 M	8.19 M	8.19 M	8.00 k	1024	1024

10 kHz Nominal Synthesis Rate (32.768 kHz oscillator reference)									
DAC Precision	**IntGenCtrl PDMCD Bit**	**Over-Sampling Factor**	**ClkSpdCtrl PLLM Register Value**	**Master Clock Rate (Hz)**	**PDM Rate (Hz)**	**CPU Clock Rate (Hz)**	**Output Sampling Rate (Hz)**	**Number of Instructs btwn DAC Interrupts**	**Number of Instructs btwn 10 kHz Interrupts**
8 bits	1	1x	0x 13	2.62 M	2.62 M	1.31 M	10.24 k	128	128
		2x	0x 26	5.11 M	5.11 M	2.56 M	19.97 k	128	256
		4x	0x 4D	10.22 M	10.22 M	5.11 M	39.94 k	128	512
		8x	0x 9B	20.45 M	20.45 M	10.22 M	79.87 k	128	1024
	0	1x	0x 26	5.11 M	2.56 M	2.56 M	9.98 k	256	256
		2x	0x 4D	10.22 M	5.11 M	5.11 M	19.97 k	256	512
		4x	0x 9B	20.45 M	10.22 M	10.22 M	39.94 k	256	1024
9 bits	1	1x	0x 26	5.11 M	5.11 M	2.56 M	9.98 k	256	256
		2x	0x 4D	10.22 M	10.22 M	5.11 M	19.97 k	256	512
		4x	0x 9B	20.45 M	20.45 M	10.22 M	39.94 k	256	1024
	0	1x	0x 4D	10.22 M	5.11 M	5.11 M	9.98 k	512	512
		2x	0x 9B	20.45 M	10.22 M	10.22 M	19.97 k	512	1024
10 bits	1	1x	0x 4D	10.22 M	10.22 M	5.11 M	9.98 k	512	512
		2x	0x 9B	20.45 M	20.45 M	10.22 M	19.97 k	512	1024
	0	1x	0x 9B	20.45 M	10.22 M	10.22 M	9.98 k	1024	1024

Chapter 5

Code Development Tools

This chapter describes the code development tools for the MSP50C6xx family of devices. The MSP50C6xx code development tool is used to compile, assemble, link, and debug programs. A reduced function C compiler, (called C––) is also part of the code development tool.

Topic **Page**

5.1	Introduction	5–2
5.2	MSP50C6xx Development Tools Guidelines	5–4
5.3	MSP50C6xx Code Development Tools	5–8
5.4	Assembler	5–11
5.5	C– – Compiler	5–16
5.6	Implementation Details	5–24
5.7	C– – Efficiency	5–37
5.8	Beware of Stack Corruption	5–57
5.9	Reported Bugs With Code Development Tool	5–58

5.1 Introduction

The MSP50C6xx code development tool is a system made up of a personal computer (PC), the EMUC6xx software, an MSP scanport interface, and a MSP50P614 connected to the application circuits.

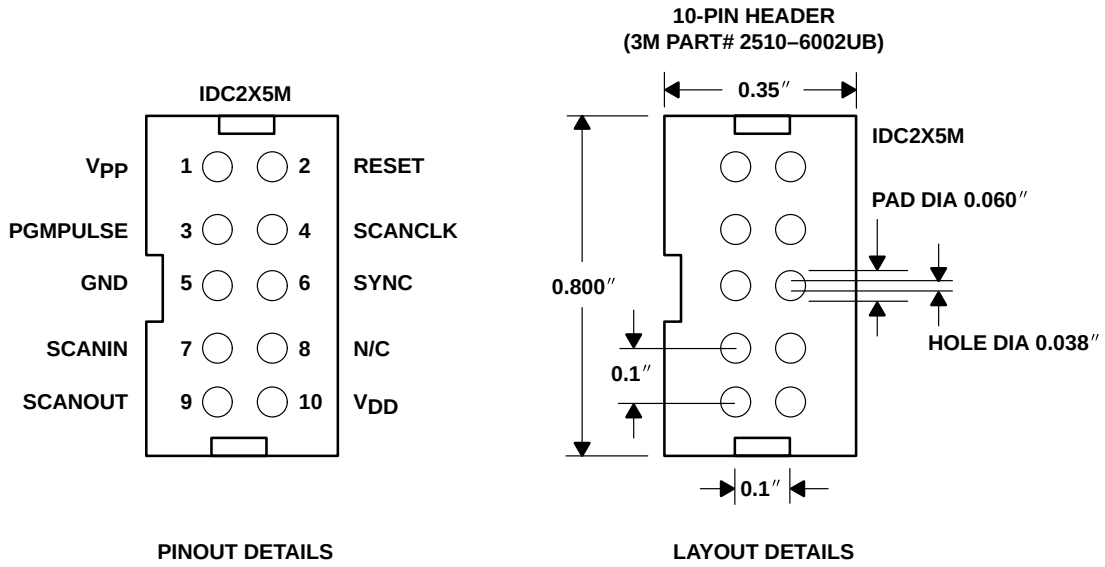
EMUC6xx is the software that executes on the PC and provides a user interface to the compiler, assembler, linker, debugger, and MSP50P614 programmer. This software gains access to the MSP50P614 and MSP50C6xx devices through a serial interface, called scanport. The MSP scanport interface (TI part number MSPSCANPORTI/F) is used to connect the scanport to an enhanced parallel port on the PC. The MSP50P614 is an EPROM based device used to emulate the MSP50C6xx devices. These EPROM based devices are packaged in a kit of 15 pieces (TI part number SDK50P614), and are only available in limited quantities to support code development.

The MSP50P614's EPROM must be programmed to debug the code in real-time. The MSP50C6xx code development tool is used to program the EPROM, set a breakpoint, and evaluate the internal registers after the breakpoint is reached. This mode is called Run Internal. The Trace mode also requires the code to be programmed into the EPROM. If a change is made to the code, the code will need to be updated and programmed into another device while erasing previous devices. This cycle of programming, debugging, and erasing typically requires several devices to be in the eraser at any time, so 10–15 devices may be required to operate efficiently.

The MSP50C6xx code development tool also supports non-real-time debugging by scanning the code sequence through the scanport without programming the EPROM. However, the rate of code execution is limited by the speed of the PC parallel port. These modes are called Run and Fast Run.

Any preproduction applications boards being used for code development must have a 13x13, 121 pin, zero insertion force (ZIF), PGA socket that allows the MSP50P614 to be easily changed. Use the PGA package pin assignments shown in Figure 7–4. These preproduction boards also have the following requirements for the development tool to function properly. (1) A 10 pin keyed IDC connector, as shown in Figure 5–1, that connects the MSP50P614 to the MSP scanport interface should be provided. (2) The VPP pin of the MSP50P614 must be pulled up with a diode connected to VDD, so the development tool can apply 12 V to this pin. (3) The development tool must be allowed to toggle the RESET pin without being loaded by any low impedance reset circuit. This can be accomplished by inserting a 1-kΩ resistor between the reset circuit and the RESET pin, and connecting the MSP scanport interface reset signal directly to the RESET pin. See the reset circuit shown in Figure 1–3.

Figure 5–1. 10-Pin IDC Connector (top view looking at the board)



It is also recommended that all production boards provide a method for connecting the MSP50C6xx code development tool to the scanport. This allows the development tool to facilitate any post-production debugging. There are several options for providing access to the scanport. If the production board has enough room, layout the footprint of the recommended connector and connect it to the scanport. The connector could be added as need for debugging. If the production board does not have enough room for the connector, put test points for the scanport signals and a connector can be hand wired to these test points. If the production boards use chip-on-board (COB), be sure to bond-out the scanport signals. It would also be helpful to layout the board so that a 1-kΩ resistor could be added in series with the reset circuit, as described in requirement (3) for the preproduction board. This resistor would not be added during production, and would be shorted with a jumper or etch on a surface layer of the board. The jumper could be removed or etch could be cut, and the resistor added when needed.

TI has two evaluation systems that may be used to develop code. The EVA50C605 and the SPEECH–EVM (requires the appropriate personality card) are basic target boards. The EVA50C605 has the minimum circuits required for supporting code development. It has a socket for the MSP50P614, a socket for a 4M bit EPROM, a reset circuit, test points for power, DAC, and I/O ports, the external oscillator and PLL filter components, and the scanport connector. The SPEECH–EVM is a generic board that supports several TI speech devices by accepting different personality cards. This board has the same features as the EVA50C605, plus a battery holder, two different speaker

amplifiers, an 8-position DIP switch and two momentary switches connected to I/O pins. These boards are discussed more in Sections 5.2.2 and 5.2.4.

5.2 MSP50C6xx Development Tools Guidelines

This is a summary of the tools needed for code development and speech editing for the MSP50C6xx family of speech processors (MSP50C614, MSP50C605, MSP50C601, and MSP50C604).

5.2.1 Categories of MSP50Cxx Development Tools

There are two kinds of tools:

- Code development tools. These are hardware and software tools for compiling, assembling, linking, and debugging code for the MSP50C6xx devices
- Speech editing tools. These are the hardware and software tools for analyzing speech files, editing speech data, and generating coded speech.

5.2.1.1 Code Development Tools

- If the user is developing code for an MSP50C604 (being used in master mode), MSP50C601, MSP50C605, or MSP50C614, the following tools are needed:
  - Hardware
    - MSPSCANPORTI/F
    - SDK50P614 (kit of 15 MSP50P614s)
    - SPEEC-EVM† or EVA50C605†‡
    - EPC50C605†
  - Software
    - MSP50C6xx code development software (EMUC6xx)
- If the user is developing code for an MSP50C604 being used in slave mode, the following tools are needed:
  - Hardware
    - MSPSCANPORTI/F
    - SDK50P614 (kit of 15 MSP50P614s)
    - SPEECH-EVM
    - EPC50C604
  - Software
    - MSP50C6xx code development software (EMUC6xx)

- If the user is developing host code to be used with a catalog MSP50C604 operating in slave mode:
  - Hardware
    - Catalog device
    - SPEECH-EVM†
    - PC50C604†

† These items are not needed if the customer designs their own preproduction application boards.
‡ Speech-EVM and EVA50C605 have similar functionality. They both function as basic target boards that support code development. For more information about these boards refer to Section 5.2.2.

5.2.1.2 Speech Editing Tools

For editing and analyzing speech for the MSP50C6xx family the following is needed:

- Hardware
  - SDS-6000
- Software
  - SDS-6000 speech editing software

5.2.2 Tools Definitions

5.2.2.1 Hardware Tools Definitions

Note:

All the following TI part numbers can be purchased through authorized TI distributors (see **http://www.ti.com/sc/docs/general/distrib.htm**).

Please contact TI speech applications group (email: **Speak2Me@list.ti.com**) for the latest version of the software.

- MSPSCANPORTI/F

  The MSP scanport interface board connects the PC's parallel port to the MSP50P614 or MSP50C6xx scanport. The user must provide a way of connecting the MSP scanport interface to their application board. See Section 5.1 for more details about this requirement.

- SDK50P614

  This is a software developers kit that contains 15 units of MSP50P614s (EPROM devices). The customer will need to have access to an EPROM eraser (not supplied by TI) to erase these devices.

- EPC50C605

The emulation personality card, for the speech-EVM, that supports code development on the MSP50C614, MSP50C605, MSP50C601, and MSP50C604 (being used in master mode). A MSP50P614 is used on this board to emulate the MSP50C6xx core. An EPROM is used on the SPEECH-EVM board to emulate the data ROM of the MSP50C601 and the MSP50C605.

- EPC50C604

  The emulation personality card, for the SPEECH-EVM, that supports code development on the MSP50C604 (being used in slave mode). A MSP50P614 is used on this board to emulate the MSP50C6xx core and the external logic devices that are built on the personality card emulate the slave mode of MSP50C604. The board has a 25-pin connector that allows a PC parallel port to emulate the host processor.

- PC50C604

  The personality card, for the SPEECH-EVM, that has a 64-pin QFP socket for a catalog MSP50C604 and a 16-pin DIP socket for a catalog MSP53C39x (see the following note). This board can be used to develop host codes for use with either a MSP50C691 or MSP53C392 slave device. It also can be used with the SDS3000 software, which is a MSP50C3x speech editing system.

Note:

The MSP50C691 and the MSP53C392 are catalog slave speech synthesizers in the MSP50C6xx and the MSP50C3x family of speech devices.

- SPEECH–EVM (see the following note)

  This board, along with the appropriate personality card, provides a basic target board that a customer can use to begin code development. The SPEECH–EVM can be used with the following personality cards:
  - EPC50C605
  - EPC50C604
  - PC50C604

  This board supports the following speaker drive options:
  - LM386 (with volume control)
  - H bridge
  - direct drive

  There is a socket for an EPROM on the SPEECH–EVM to emulate the DATA ROM in the MSP50C601 and the MSP50C605.

- EVA50C605 (see the following note)

  Same as SPEECH-EVM.

Note:

The SPEECH-EVM and EVA50C605 have similar functionality. They both function as basic target boards that support code development. One of the differences is that the SPEECH-EVM has a battery holder, and the EVA50C605 does not. The SPEECH-EVM also has the hardware circuits to drive an 8-Ω speaker using the LM386 or H-bridge option. However, the EVA50C605 can only be used with a 32-Ω speaker (direct drive).

- SDS-6000

  The hardware works with SDS6000 software to allow speech editing as well as verification of speech quality through a MSP50x6xx device. It connects to a PC through a parallel port.

5.2.2.2 Software Tools Definitions

- MSP50C6xx code development software (EMUC6xx)

  The PC based software is used for MSP50C6xx code development and requires Microsoft Windows 95™ or 98™ operating systems. It is one part of the MSP50C6xx code development tools, along with the MSP scanport interface, and the MSP50C6xx device on an application board.

- TITALKS.zip (formerly known as FIXEDxx.zip)

  This contains the latest version of TI compression algorithms. The file TITALKS.ZIP contains the base code for the MSP50C6xx, which includes all the TI coders (MELP, CELP, LPC, and ADPCM). There are some sample codes for LCD drivers, timer 1, and timer 2 interrupts etc. This software provides a good starting point for the customer to develop the code for MSP50C6xx devices. Examples of RAM overlay methods have been included for the customers' benefit.

- SDS6000

  This software is used for speech editing and is designed to be used with the SDS-6000 hardware.

Note:

Please contact TI Speech Applications Group (email: **Speak2Me@list.ti.com** for the latest version of the software.

5.2.3 Documentation

- ❑ MSP50C6xx Product Folders

  http://www.ti.com/sc/docs/products/speechh/index.htm

- ❑ MSP50C6xx User's Guide

- ❑ Datasheet

  MSP50C614:

  MSP50C605:

  MSP50C601:

  MSP50C604:

- ❑ Applications Notes

  Documents that help users in developing code for MSP50C6xx devices are available.

- ❑ SDS6000 Speech Editing Tool manual

- ❑ Schematics

  Reference designs/schematics for the daughter cards. Schematics of the SPEECH–EVM and the EVA50C605 are also available.

5.3 MSP50C6xx Code Development Tools

5.3.1 System Requirements

- ❑ PC with Intel 486™ or Pentium™ class processor
- ❑ Microsoft Windows 95™, or Windows 98™ operating system
- ❑ 16M-Byte memory
- ❑ 8M-Byte hard disk space
- ❑ Enhanced parallel port interface

5.3.2 Hardware Tools Setup

Step 1: Plug in an appropriate personality card (see the following note) on the SPEECH-EVM or EVA50C605.

Note:

EPC50C605: developing code for MSP50C604 (in master mode, MSP50C601, MSP50C605, or MSP50C614).

EPC50C604: developing code for a custom MSP50C604 used in slave mode.

PC50C604: developing host code to be used with a catalog MSP50C604 (slave mode) device.

Step 2: Connect a speaker (see the following note) to the SPEECH-EVM or EVA50C605 board.

Note:

The SPEECH-EVM or EVA50C605 supports following speaker drive options:

- ❑ LM386 (with volume control)
- ❑ H-bridge
- ❑ Direct drive

If you choose LM386 or H-bridge as the speaker drive option, you have to use a 8-Ω speaker. If you choose direct drive as the speaker drive option, you have to use a 32-Ω speaker.

Step 3: Use the provided parallel cable to connect the PC's parallel port and scanport interface.

Step 4: Connect the scanport interface to the SPEECH-EVM or EVA50C605.

Step 5: Connect the scanport interface to a power supply. The red light on the scanport interface should be ON.

Step 6: Place a MSP50P614 device on the personality card that you use in Step 1.

Step 7: Apply power to SPEECH-EVM (see the following note) or EVA50C605. The green light on the scanport interface should be ON.

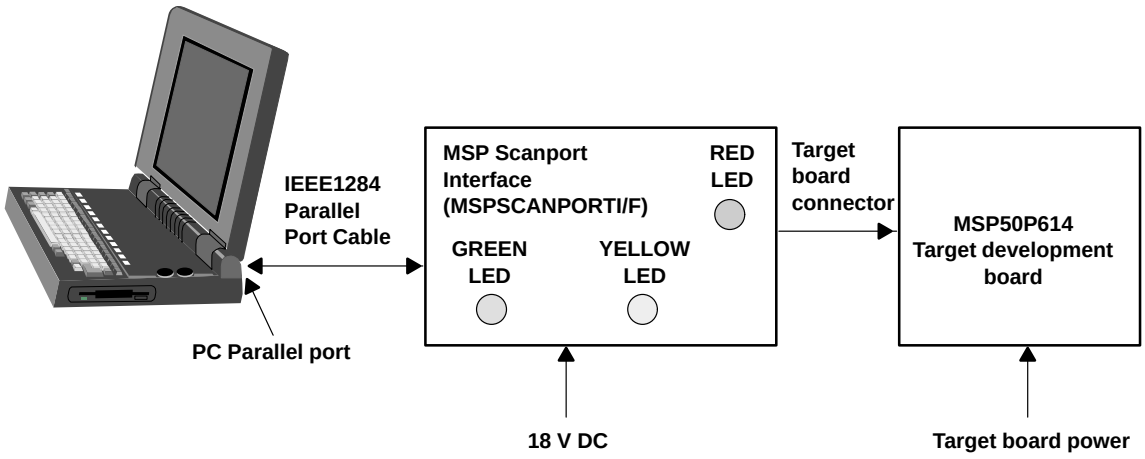
Note:

There is a three-way switch at the edge of the SPEECH-EVM board. After you apply power to the SPEECH-EVM, you have to turn on the SPEECH-EVM. There are two ways to turn on the board depending on the power sources:

- If you are using the on board with AAA batteries as the power source, you have to slide the switch to the BATT position to turn on the board.
- If the power is provided externally from TB1 connector, you have to slide the switch to the EXT position to turn on the board.

Step 8: Open EMU50C6xx software. The yellow light on the scanport interface should be ON.

Figure 5–2. Hardware Tools Setup



LED DESCRIPTION	
Red	MSPSCANPORTI/F power
Yellow	Emulation mode/programming (Emul/Prog)
Green	Target board power

5.4 Assembler

5.4.1 Assembler Directives

Assembler directives are texts which have special meaning to the assembler. Some of these directives are extremely helpful during conditional compiling, debugging, adding additional features to existing codes, multiple hardware development, code release etc. Other directives are an essential part of the assembler to initialize variables with values, assigning symbols to memory locations, assigning origin of a program, etc. The assembler directives that start with a # (hash) sign cannot have spaces before the directive. The following assembler directives are recognized by the assembler. Some of these assembler directives use **expressions** and **symbols**. These are explained below:

expression can be any numeric value. Addition, subtraction, and multiplication are allowed.

Examples:
`(128 / 2 ) * 2 + (220 / 5) + 2 + *0x200` equates to 0xAE + *0x200, where *0x200 indicates data memory location.

`(2 * 2 / 2 + ((5 * 2) * 3) / 2) | (0x0F & 0x04)` equates to 0x15. Note that bitwise AND (& operator) and OR (| operator) operations are allowed.

`(10 * 2) + 5 * *0x120` expression points to data memory content at 0x120, multiplies decimal 5 to it, and finally adds decimal 20. Note that a space is required between successive asterisks (*). Also note that *0x120 indicates content of memory location at 0x120 hex.

The grammar for expressions and symbols are as follows:

```
number:      number| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
expression:  number
             | expression + expression
             | expression – expression
             | expression * expression
             | expression / expression
             | expression | expression
             | expression & expression
             | ~expression
             | –expression
             | +expression
             | *expression
```

| **(***expression***)**

(~ indicates bitwise complement)

symbol is any alphanumeric text starting with an alphabetic character, a number, or an expression.

Examples:

```
SYM1  EQU (12 * 256)
SYM2  EQU SYM1 * (32 / 4)
SYM3  EQU SYM1 * SYM2 – *0x200
```

From the above example SYM1, SYM2 and SYM3 are symbols for some expression. The grammar for a Symbol is as follows:

```
symbol:        expression
               | symbol
```

Expression Restrictions: It is recommended that a space be inserted between the operator (i.e., +, –, *, /, |, &) and the symbol or numeric expression to perform arithmetic and bitwise operations. For example `ADD A0, A0, 1 + –2`, adds a –1 to **A0**, because the argument is read as 1+(–2) = –1; but writing the argument as 1+–2 may or may not give the correct result. Outside parenthesis are not allowed in instruction arguments. For example, `ADD A0~,A0~,(1 + (2 * 300) – 256)` causes a compile time syntax error. But removing the outside parenthesis i.e., `ADD A0~,A0~, 1 + (2 * 300) – 256`, causes no error.

#ELSE: see **#IF** and **#IFDEF**

#END_FT : This directive is created by the C– – compiler when it outputs assembly code to a file. It marks the end of the *function table* used to track function calls and C– – variables in the emulator. ***Users should NEVER use this directive in an assembly language program***.

#ENDIF: marks the end of a conditional assembly structure started by **#IF** or **#IFDEF**

#IF expression: The start of a conditional assembly structure expression is an arithmetic expression that can contain symbols. ***Caution: since conditional assembly is resolved during the first pass of the assembler, no forward referenced symbols should be used in a conditional assembly expression.*** If an expression is TRUE (non zero), then the lines following this directive are assembled until a **#ELSE** or a **#ENDIF** directive is encountered. If an expression is FALSE (equal to zero), then all input lines are skipped until a **#ELSE** or a **#ENDIF** directive is encountered. If a **#ELSE** directive is encountered first, all lines following it are assembled, until a **#ENDIF** directive is found.

Example:

#IF expression
; do something here
#ELSE
; do other things here
#ENDIF

#IFDEF symbol: Start of a conditional assembly structure. If the symbol has been defined (either with a **#DEFINE** directive or an EQU directive) then the lines following this directive are assembled until a **#ELSE** or a **#ENDIF** directive are encountered. If symbol has not been defined, then all input lines are skipped until a **#ELSE** or a **#ENDIF** directive is encountered. If a **#ELSE** directive is encountered first, all lines following it are assembled, until a **#ENDIF** directive is found.

#IFNDEF: Start of a conditional assembly structure. If symbol has NOT been defined then the lines following this directive are assembled until a **#ELSE** or a **#ENDIF** directive is encountered. If symbol has been defined (either with a **#DEFINE** directive or an EQU directive), then all input lines are skipped until a **#ELSE** or a **#ENDIF** directive are encountered. If a **#ELSE** directive is encountered first, all lines following it are assembled, until a **#ENDIF** directive is found.

Example:

#IFDEF symbol
; do something here
#ELSE
; do other things here
#ENDIF

#IFNDEF symbol
; do something here
#ELSE
; do other things here
#ENDIF

#START_FT: This directive is created by the C– – compiler when it outputs assembly code to a file. It marks the beginning of the *function table* used to track function calls and C– – variables in the emulator. ***Users should NEVER use this directive in an assembly language program.***

AORG expression: Marks the start of an ABSOLUTE segment code, i.e., a segment that cannot be relocated by the linker. *expression* evaluates to the starting address of the absolute segment in the program memory.

BYTE expression[,expression]: Introduces one or more data items, of BYTE size (8 bits) . The bytes are placed in program memory in the order in which they are declared.

CHIP_TYPE chip_name: This directive is provided for compatibility with future chips in the same family. It defines chip parameters (such as RAM and ROM size) for the assembler. For now, the only defined chip name is MSP50C6xx.

DATA expression[,expression]: Introduces one or more data items, of WORD size (16 bits) . The words are placed in the program memory in the order in which they are declared. Even though the program memory is 17 bits wide, only 16 bits can be read using assembly instructions (like **MOV A0,*A0**), so the **DATA** directive only stores 16 bits per data expression.

DB expression[,expression]: Equivalent to **BYTE** directive

DEF symbol[,symbol]: Equivalent to **GLOBAL** directive

DW expression[,expression]: Equivalent to **DATA** directive

END expression: Expression defines the start vector for the current assembly program. This directive generates the following assembly code;

AORG 0xFFFF

DATA expression

which defines the start vector of the program, i.e., the program address where execution begins when the chip is installed.

label EQU expression: Associates the value of *expression* with *label*.

EXTERNAL symbol[,symbol]: This directive is used to indicate to the assembler that one or more symbols are external references, i.e., symbols that will be resolved by the linker.

GLOBAL symbol[,symbol]: This directive is used to indicate to the assembler that one or more symbols are global references. These symbols MUST be defined in the current file, and will be used by the linker to resolve external references (present in other files). **GLOBAL** should only be used for PROGRAM labels. RAM variables are handled with the **GLOBAL_VAR** directive.

GLOBAL_VAR symbol[,symbol]: This directive allows a RAM variable to be referenced from another file. **GLOBAL_VAR** should be used prior to defining a RAM variable (with the RESW directive, for example). The file that references the variable should declare it as EXTERNAL (of REF). Note that this technique can also be used to make constants defined with the EQU statement available to other files.

INCLUDE *filename*: This directive is used to insert another file in the current assembly file. The name of the file must be enclosed in double quotes. If the file name itself is enclosed in angled brackets (<>), then the assembler will first look for the include file in the include directory list that is passed as an argument during the DLL call.

LIST: The lines following this directive are included in the listing file (extension .lst) created by the assembler.

REF symbol[,symbol]: Equivalent to EXTERNAL directive

label RESB expression: This directive is used to reserve the number of bytes indicated by expression, starting at the current RAM address. Label is given the value of the current RAM address.

label RESW expression: This directive is used to reserve the number of words indicated by expression, starting at the current RAM address. label is given the value of the current RAM address. If the current RAM address is not EVEN, the assembler increments it by 1 before allocating the desired amount. (Note that RAM locations are accessed by their BYTE address in MSP50C6xx assembly language, i.e., word 1 is at address 2, etc...)

RORG expression: Marks the start of a RELATIVE segment code, i.e., a segment that can be relocated by the linker. Expression is an arbitrary number, but it must be present or an assembly error will occur.

STRING *text_string*: Equivalent to the **TEXT** directive, but the text is terminated by a 0. (automatically done by the assembler)

TEXT *text_string*: Equivalent to the **BYTE** directive, but the data is a text string enclosed in double quotes.

UNLIST: The lines following this directive are not included in the listing file (extension .lst) created by the assembler.

5.5 C– – Compiler

The C– – compiler generates an assembly language file of the same name, with extension `.opt`. It also generates a file with extension `.glb` where global variable initialization is taken care of, if the routine *main* was encountered in the current file. A file with extension `.ext` is also generated to take care of global and external declarations that will be used by the assembler. These two files are included in the `.opt` file generated by the C– – compiler. Note that all symbols defined in C– – source code are changed before being written to assembly language: an underscore character is put in front of the first character of each symbol. Also note that local labels created by the C– – compiler are built using the current source file name followed by an ordinal number. Consequently, to avoid problems at link time due to symbols bearing the same name, never use symbol names starting with an underscore in assembly language files. It is ***imperative*** to use file names that are different for C– – files (extension `.cmm`) and assembly language files (extension `.asm`).

5.5.1 Foreword

C–– is a *high level* language to be used with the MSP50C6xx microprocessors. Although it looks a lot like C, it has some limitations/restrictions which will be highlighted throughout the remainder of this chapter. This language is compiled into MSP50C6xx assembly language.

5.5.2 Variable Types

Type Name	Mnemonic	Range	Size in Bytes	Example
Integer	int	[–32768,32767]	2	int i,j;
Character	char	[0,255]	1	char c,d;
Array of integer	int	Not Applicable	Not Applicable	int array[12];
Array of characters	char	Not Applicable	forced to even	char text[20]
Pointer to integer	int *	Not Applicable	2	int *j;
Pointer to character	char *	Not Applicable	2	char *string;

Notes:
1) There is a major difference between an MSP50C6xx integer string and an array of integers: an array of integers is an ordered set of *n 16 bit integers*, whereas an integer string of length *n* represents a *single* integer with 16**n* bits. In C– –, MSP50C6xx strings are declared as arrays of integers, but must be operated upon using the special purpose string arithmetic functions described below.
2) As in regular C, the above types can be qualified with the word unsigned.
3) There is another important qualifier that is special to C– – : *constant*. We made the mnemonic purposely different from the usual C *const* qualifier, because it is not exactly equivalent. It is used to initialize arrays in program ROM. A good use of it would be for a sine table, for example. The syntax is simple, for example:
   constant int array[10]={1,2,3,4,5,6,7,8,9,10},dummy;
4) will create a series of DATA statements in the assembly language output file. Uninitialized constants (like *dummy* above) generate a warning and are initialized to zero. Constants are to be handled with care. Since they cannot be accessed the same way as RAM variables, special purpose functions have to be used to utilize constants in a program. The most general of these functions is xfer_const, which transfers values from the program ROM to the RAM. Also, constants MUST BE GLOBAL. Do not pass a constant as an argument.
5) The common C types float, struct, union and long are not implemented. (Note that *long* is a subset of *string of integer*).

5.5.3 External References

All RAM allocations in the assembler are global. This results in the following implications for C– – variables:

- ❑ Only the file containing the main routine can contain global variable definitions.
- ❑ Global variables referenced in other files must have been declared as external (keyword extern) at the beginning of the file.
- ❑ A function referenced in a file but not defined in that same file must be introduced with a function prototype in the file where it is referenced (no need for the extern keyword).

5.5.4 C– – Directives

C– – has a limited number of directives and some additional directives not found in ANSI C compilers. The following directives are recognized by the compiler.

5.5.4.1 #define

This directive is used to introduce 2 types of macros, in typical C fashion:

Without Arguments:
defines a replacement string for a given string

Example:
```
#define PI 3.1415926535
```

Every occurrence of the token PI will henceforth be replaced with the string 3.1415926535.

If there is no replacement string, the given string is deemed *defined*: this can be used in conjunction with the **#ifdef** / **#ifndef** directives. It is also possible to *undefine* a macro with the #*undefine* directive.

With Arguments:
The macro name must be **immediately** followed by a pair of parenthesis, which introduces the arguments. This is completely compatible with the usual C definition.

Example:
#define modulo(i,j) (i%j)

Every occurrence of the word modulo followed by an expression in parentheses will be replaced by (i%j), where i is the first argument in the parenthesis, and j the second argument. modulo((a*b),c) will thus be replaced by ((a*b)%c).

5.5.4.2 #undefine

The string following this directive is removed from the list of macros. There is no warning if the string is not found in the macro list.

5.5.4.3 #include

As in regular C, this directive allows for the insertion of a file into the current file. If the file name that follows is enclosed in < >, the system searches the include directories for the file, otherwise, if it is enclosed in " ", the current directory is searched.

Example:
#include “file.h”
#include <stdio.h>

The include directories are defined on the `cmm_input` structure passed to the compiler. There is no limit to the nesting of include files.

5.5.4.4 #asm

All text following this directive is inserted *as is* in the output file, and is considered as assembly language (hence not compiled). The insertion continues until a **#endasm** directive is found. Note that both #asm and **#endasm** must be at the beginning of a line, and that all text following them on the same line is ignored.

5.5.4.5 #endasm

Signals the end of assembly language insertion. Must be paired with a **#asm** directive.

5.5.4.6 #ifdef, (#ifndef)

Starts conditional assembly if token following it has been defined (not been defined) by a **#define** directive. These directives are terminated by a **#endif** directive, and can be coupled with a **#else** directive, as in regular C. Note that the test can only check if the named token is currently defined or undefined.

5.5.4.7 #if

Starts conditional assembly if the expression following it evaluates to a non zero value. This directive is terminated by a **#endif** directive, and can be coupled with a **#else** directive, as in regular C.

5.5.4.8 #else

See **#if** directive.

5.5.4.9 #endif

Must be present to terminate a **#ifdef** or **#ifndef** directive

Note:

Typedef is not supported in C– –.

5.5.5 Include Files

There are currently two include files supplied with C– –, `cmm_func.h`, which contains function prototypes for the C– –functions and `cmm_macr.h` which contains some predefined macros. Both files are listed below:

```
/*******************************/
/* Prototypes for C– –functions */
/*******************************/
cmm_func add_string(int *result,int *str1,int *str2,int lg);
cmm_func sub_string(int *result,int *str1,int *str2,int lg);
cmm_func mul_string(int *result,int *str1,int mult,int lg1,int lgr);
cmm_func umul_string(int *result,int *str1,unsigned int mult,int lg1,int lgr);
cmm_func or_string(int *result,int *str1,int *str2,int lg);
cmm_func and_string(int *result,int *str1,int *str2,int lg);
cmm_func xor_string(int *result,int *str1,int *str2,int lg);
cmm_func not_string(int *result,int *str1,int lg);
cmm_func neg_string(int *result,int *str1,int lg);
cmm_func copy_string(int *output,int *input,int lg);
cmm_func rshift_string(int *output,int *input,int rshift,int lg);
#ifdef _CMM
cmm_func strcpy(char *outstring,char *instring);
cmm_func strlen(char *instring);
cmm_func calloc(int nitems,int size);
cmm_func malloc(int size);
cmm_func free(int *ptr);
#endif
cmm_func test_string(int *string1,int *string2,int lg,int oper);
cmm_func xfer_const(int *out,int *cst_addr,int lg);
cmm_func xfer_single(int *out,int *cst_addr);
/*******************************/
```

Note the requirement that C– – function declarations (including *main*) be preceded by the keyword *cmm_func*. Also note the conditional assembly portion, used for compatibility with Borland C.

```
/******************/
/* Macros for C– – */
/******************/

#define STR_LENGTH(i) (i–2)

/*****************/
```

Major Differences between C and C– –

Although we have tried to keep the differences between *regular* C and C– – to a minimum, there are still a few that require explanation.

5.5.6 Function Prototypes and Declarations

C– – function prototypes and declarations MUST be preceded with the keyword *cmm_func.*

Since all functions return through accumulator A0, all functions are of type integer. The function type may be omitted in the function declaration. If present, it is ignored anyway. Trying to typecast a function as returning a pointer will result in a compiler error.

Note: To change a C– – program back into a regular C program (at least from the point of view of function prototypes and declarations), the following line can be inserted at the beginning of the C– –program:

```
#define cmm_func
```

A library of regular C functions to substitute for the special MSP50C6xx functions is supplied with the C– – compiler, allowing the user to compare the results of regular C programs with those of C– –programs. The library is contained in the C source file `cmm_func.c` .It should be linked with the C equivalent of the C– – program, and run in Borland C.

Note:

To use external functions in C– –, a function prototype should be placed in the file that calls the external function.

5.5.7 Initializations

Due (in part) to the architecture of the MSP50C6xx processors, initialization is only allowed for global variables. As a side effect, local static variables are not allowed. For example, a global array can be declared and initialized as follows:

```
int int_array[5]={1,2,3,4,5};
```

Initialization values are stored in program memory.

5.5.8 RAM Usage

RAM location 0 is reserved (and used intensively) by the compiler. The choice of location 0 does not conflict with the usual definition of a NULL pointer.

5.5.9 String Functions

Arithmetic string functions are special functions that perform string arithmetic. The functions currently implemented are shown in Table 5–1.

Table 5–1. String Functions

add_string(int *result,int *str1,int *str2,int lg) adds strings str1 and str2, of length lg (+2), and puts the result in string result
sub_string(int *result,int *str1,int *str2,int lg) subtracts strings str2 from str1, of length lg (+2), and puts the result in string result.
mul_string(int *result,int *str1,int mult,int lg1,int lgr) multiplies string str1 of length lg1 (+2) by integer multiple, and puts the result in string result, of length lgr (+2).
umul_string(int *result,int *str1,int mult,int lg1,int lgr) same as previous one, with UNSIGNED multiply
or_string(int *result,int *str1,int *str2,int lg) ors strings str1 and str2, of length lg (+2), and puts the result in string result.
and_string(int *result,int *str1,int *str2,int lg) ands strings str1 and str2, of length lg (+2), and puts the result in string result.
xor_string(int *result,int *str1,int *str2,int lg) exclusive ors strings str1 and str2, of length lg (+2), and puts the result in string result.
not_string(int *result,int *str1,int lg) takes the 1's complement of string str1, of length lg (+2), and puts the result in strings result.
neg_string(int *result,int *str1,int lg) takes the 2's complement of string str1, of length lg (+2), and puts the result in strings result.
test_string(int *string1,int *string2,int lg,int oper) performs a logical test (operation) on strings string1 and string2 of length lg (+2). The logical value is returned in A0. If string2 is NULL, the logical test is performed between string string1 and a zero string.
operator can take the following values: (predefined constants) **EQS_N == ?** **NES_N !== ?** **LTS_N < ?** **LES_N <= ?** **GES_N >= ?** **GTS_N > ?** **ULTS_N < ? (unsigned)** **ULES_N <= ? (unsigned)** **UGES_N >= ? (unsigned)** **UGTS_N > ? (unsigned)**

A major feature of the MSP50C6xx is that the string length present in the string register is the actual length of the string minus two. To avoid confusion, a macro is supplied that automatically translates the real length of the string to

the MSP50C6xx length of the string. It is included in the `cmm_macr.h` file, and is called **STR_LENGTH**(lstr). For example, **STR_LENGTH**(8) is 8–2 = 6.

Also note that the user has to supply the length of the input string and the length of the output string in the string multiply operations: the result of multiplying a string by an integer can be one word longer than the input string. Unpredictable results may occur if parameter lgr is not at least equal to lgr+1.

5.5.10 Constant Functions

The only two *constant* functions implemented in C– – are `xfer_const` and `xfer_single`.

```
cmm_func xfer_const(int *out,int *constant_in,int lg)
```

It transfers `lg`+2 integers from program ROM starting at address `constant_in` to RAM, starting at address `out`. Note that `constant_in` is not doubled, because it is used in **A0** in a **MOV A0,*A0** operation. The C– – compiler takes care of this.

```
cmm_func xfer_single(int *out,int *constant_in)
transfers a single value.
```

An example of the use of `xfer_const` is:

```
int array[8],i;
const int atan[80*8] ={.........640 integers );
/* .... */
for(i=0;i<80;i++){
xfer_const(array,&atan[i*8],STR_LENGTH(8));
/* ... now use array normally
..... */
}
```

5.6 Implementation Details

This section is C– – specific.

5.6.1 Comparisons

We use the **CMP** instruction for both signed and unsigned comparisons. The two integers a and b to be compared are in **A0** and **A0~**.

CMP A0,A0~ : **A0** contains a, **A0~** contains b

A0	A0~	ACO	AZ	ANEG
5	0	1	0	0
5	1	1	0	0
0	5	0	0	1
1	5	0	0	1
0	0	1	1	0
5	5	1	1	0
FFFF	0	1	0	1
0	FFFF	0	0	0
FFFF	FFFF	1	1	0
FFFF	FFFE	1	0	0
FFFE	FFFF	0	0	1

❑ Signed comparison of a and b. (a is in **A0**, b is in **A0~**)

Assembly	Test	Condition
_eq	a = b	AEQ
_ne	a != b	!AEQ
_lt	a < b	ALZ
_le	a <= b	!AGT
_ge	a >= b	!ALZ
_gt	a > b	AGT

❑ Unsigned comparison of a and b. (a is in A0, b is in A0~)

Assembly	Test	Condition
_ult	a < b	AULT
_ule	a <= b	!AUGT
_uge	a >= b	!AULT
_ugt	a > b	AUGT

The small number of comparisons was an invitation to use them as vector calls. We return a 1 or 0 in **A0** as the result of the comparison, and also set flag 2 if the comparison is true. The flag is not currently used by the compiler.

It is important to note that functions return their results via **A0**, but there is no guarantee that the *absolute* value of the **A0** pointer is not changed by the function. To compare integers a and b: after loading a in **A0**, and b in **A0~**, do a vector call to the appropriate comparison routine:

Assembly	Vector
_eq	0
_ne	1
_lt	2
_le	3
_ge	4
_gt	5
_ult	6
_ule	7
_uge	8
_ugt	9
_lneg	10

We return the result of the comparison in Flag 2 (set for TRUE, reset for FALSE), and in A0 (1 for TRUE, 0 for FALSE). We have also implemented vector calls for string comparisons. There are a few C callable routines that make use of those calls. (`test_string`, `or_string`, `and_string`, `xor_string`, `neg_string`, `not_string`)

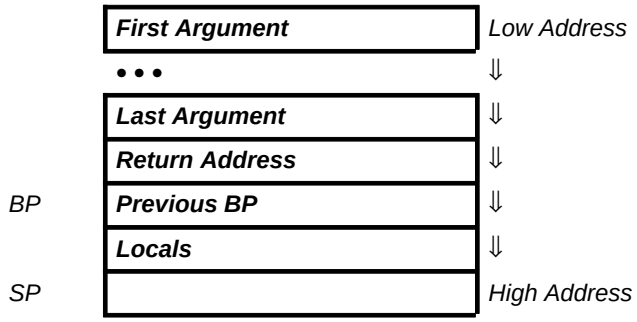
5.6.2 Division

Integer division currently requires the use of several accumulator pointers. We divide a 16 bit integer located in **A0** by a 16 bit integer located in **A0~**. We return the quotient in **A0~**, and the remainder in **A0**. We make use of **A3~** and **A3** for scratch pads. We also set flag 1 if a division by zero is attempted, and zero out the quotient and the remainder in this case. We also use **PH** for temporary storage of the divisor.

5.6.3 Function Calls

Every function is associated with a stack frame. A regular C program is initially given control by a call to `main()`. A C– – program starts with a jump to the `_main` symbol, which must therefore be present in the C– – source code.

The stack frame has the following structure:



BP is the frame pointer (base pointer), **SP** the stack pointer.

We use **R7** for stack pointer, and yet another register for **BP**, REG_BP (**R5**, because of its special arithmetic capabilities). Before a function is called, the arguments are pushed on the stack, first argument first. The function call automatically pushes the return address on the stack. Immediately upon entering the function body, the current **BP** is pushed on the stack to preserve it, so that the stack pointer now points to the next location. This location is copied to REG_BP, which becomes our fixed reference point for the current function. Locals are then allocated on the stack from this starting location.

When the function returns, **SP** is made to point to the return address, after the previous BP is popped. The return is performed by a **RET** instruction. The calling routine is then responsible for moving the stack pointer to its previous location, before the arguments were put on the stack. Because all functions return via **A0**, the **only** function return type allowed is integer. Our implementation of C– – allows for function prototyping, and checks that prototype functions are called with the correct number of arguments. Function

declarations (or function prototypes) are introduced by the mnemonic **`cmm_func.`** We only allow the new style of function declarations /prototypes, where the type of the arguments is declared within the function's parentheses. For example:

`cmm_func bidon(int i1,char *i2)` is valid, but:

`cmm_func bidon(i1,i2) int i1,char *i2;` is invalid.

Note: The exact implementation of the MSP50C6xx stack is as follows:

on CALL:

1) Increment **R7**
2) Transfer **TOS** (top of stack) register to ***R7**
3) Transfer return address to **TOS** register

on RET:

1) next **PC** = **TOS**
2) transfer ***R7** to **TOS**
3) decrement **R7**

We can freely manipulate **R7 before** a **CALL**/**C*cc*** and **after** a **RET** to load and unload arguments to and from the stack. The **TOS** register should never be altered in the body of a function.

5.6.4 Programming Example

The following example implements string multiplication (i.e., the multiplication of 2 integer strings). The same source file (with the exception of the first line) can be used for C– – or regular C. In the case of regular C, it has to be compiled and linked with `cmm_func.c`

```
#define _CMM /*must be present for C– –compiler ONLY*/
#ifdef _CMM
#else
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include “cmm_back.h”
#endif
#include “cmm_func.h”
```

```
#include “cmm_macr.h”
constant int M1[4]={0x04CB,0x71FB,0x011F,0x0};
constant int M2[4]={0x85EB,0x8FD9,0x08FB,0x0};
cmm_func string_multiply(int *p,int lgp,int *m1,int lgm1,int *m2,int lgm2)
{
/* note: length of p,(lgp+2) must be at least (lgm1+2) + (lgm2+2) +1 */
/* this function string multiplies string m1 of length lgm1+2 by string m2 of
   length lgm2+2, and puts the result into string p, of length lgp+2 */
      int sign,i,j;
      int *mm1,*mm2,*pp;
      sign=1;
      mm1=calloc(sizeof(int),lgm1+2);
      mm2=calloc(sizeof(int),lgm2+2);
      pp =calloc(sizeof(int),lgp+2);
      if(test_string(m1,0,lgm1,LTS_N))
      {
         neg_string(mm1,m1,lgm1);
         sign*=-1;
      }
      else
         copy_string(mm1,m1,lgm1);
      if(test_string(m2,0,lgm2,LTS_N))
      {
         neg_string(mm2,m2,lgm2);
         sign*=-1;
      }
      else
         copy_string(mm2,m2,lgm2);
      for(j=0;j<lgp+2;j++)
      p[j]=0;
      for(i=0;i<lgm2+2;i++)
      {
         for(j=0;j<lgp+2;j++)
         pp[j]=0;
         umul_string(&pp[i],mm1,mm2[i],lgm1);
```

```
         add_string(p,pp,p,lgm1+i+1);
      }
      if(sign == –1)
      {
         neg_string(pp,p,STR_LENGTH((lgp+2)));
         copy_string(p,pp,STR_LENGTH((lgp+2)));
      }
      free(mm1);
      free(mm2);
      free(pp);
}
cmm_func main(int argc,char *argv)
{
   int m1[4],m2[4],product[9];
   xfer_const(m1,M1,STR_LENGTH(4));
   xfer_const(m2,M2,STR_LENGTH(4));
   string_multiply(product,STR_LENGTH(9),m1,STR_LENGTH(4),m2,STR_LENGTH(4));
}
```

5.6.5 Programming Example, C –– With Assembly Routines

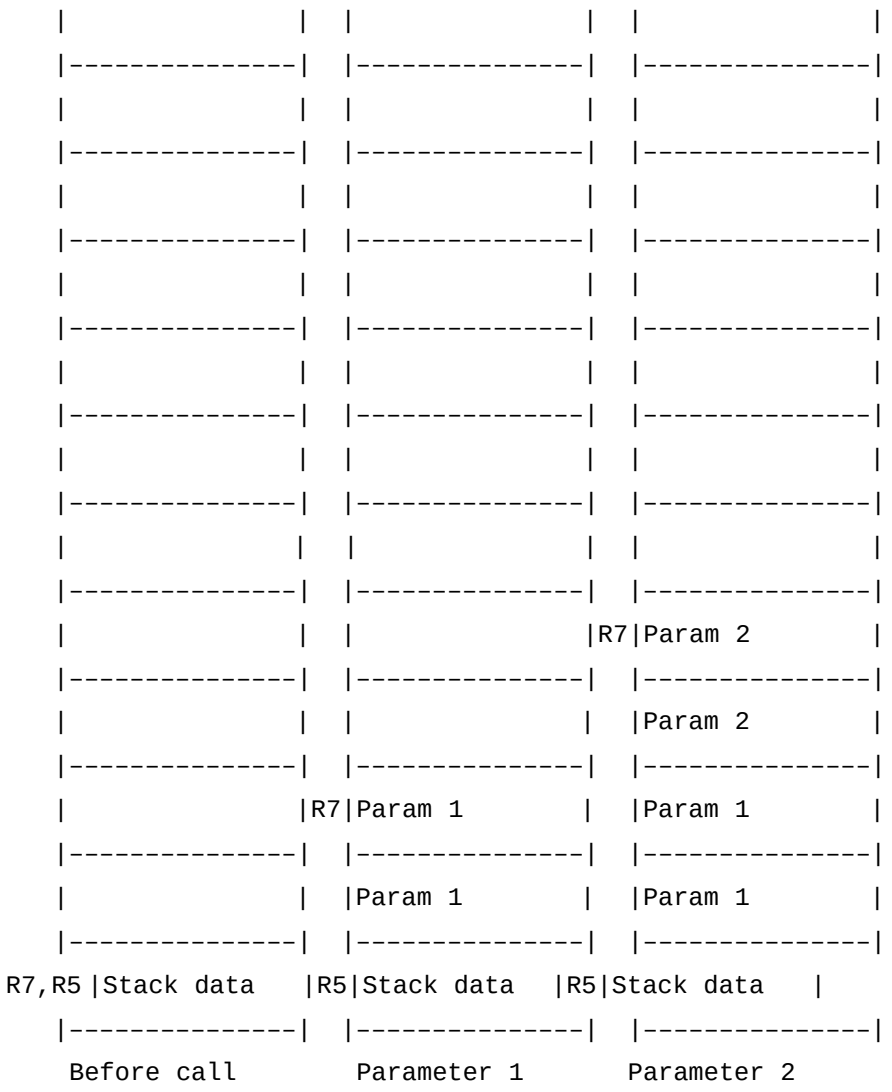
There are several important considerations when using the C–– compiler. The ram allocation must be coordinated so that a location isn't accidentally used twice. In assembly this is usually done with IRX files by making each label equal to the location of the previous one, plus whatever storage space is needed. All of the IRX files for a project are then combined in a master IRX file so that the space for each sub file can be allocated. For example (a master IRX file):

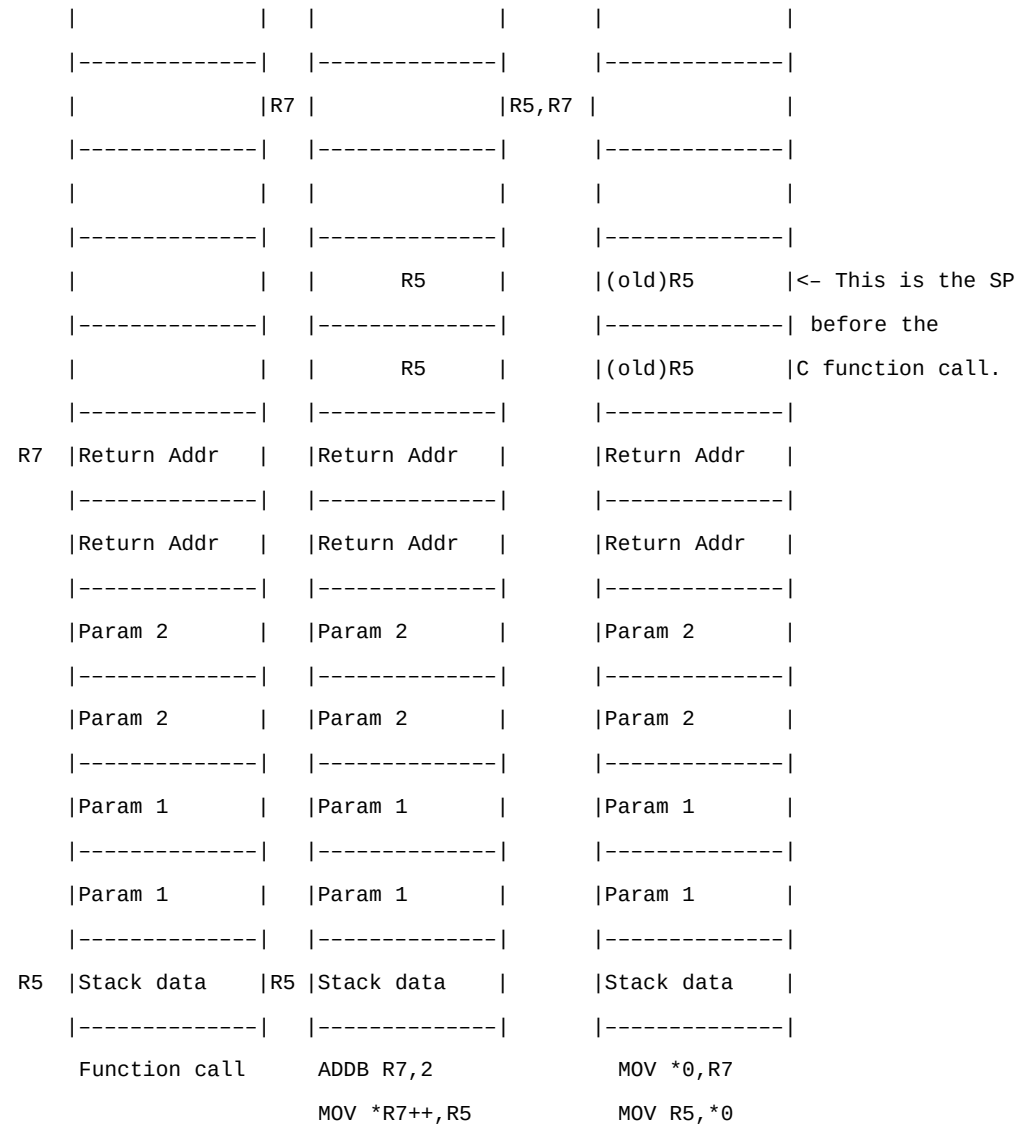
```
RAM_SIZE        equ640
STACK           equ2 * (RAM_SIZE – 14)
BEGIN_RAM    equ0
RESERVED        equ   BEGIN_RAM + 2 * 1

RAMSTART_INT equRESERVED
   include ”..\inter\inter_ram.irx”
RAMSTART_ASM equRAMEND_INT
   include      ”. .\asm_ram.irx”
```

Here the sub files are inter_ram.irx and asm_ram.irx. The allocation for inter_ram.irx begins at memory location 2. This is because the memory location 0 is reserved for use by the C–– compiler. The allocation for asm_ram.irx begins where the allocation ended for inter_ram.irx. More .irx files can be chained on in this manner, and all of the allocation is kept organized. When C–– is added to a project, it is important to make sure that the C–– variables are not allocated in locations already used by assembly variables. This is accomplished with a dummy array, bogus, located in the file ram.irx. It is simply an integer array that is included in the C–– program so that it is the first variable allocated. By making its size equivalent to the amount of memory used for assembly variables, the C–– variables that the user defines are allocated in unused memory. It can be set by building the project and finding the location of the last assembly variable. This can then be converted from hexadecimal to decimal and divided by two (because a C–– int is 16 bits) to find the correct size for bogus. Bogus can be made larger for extra safety as long as enough memory is left over for the C–– variables and the stack. If space allows, it is a good idea to add a few extra words to bogus in case assembly variables are added to the project without modifying bogus.

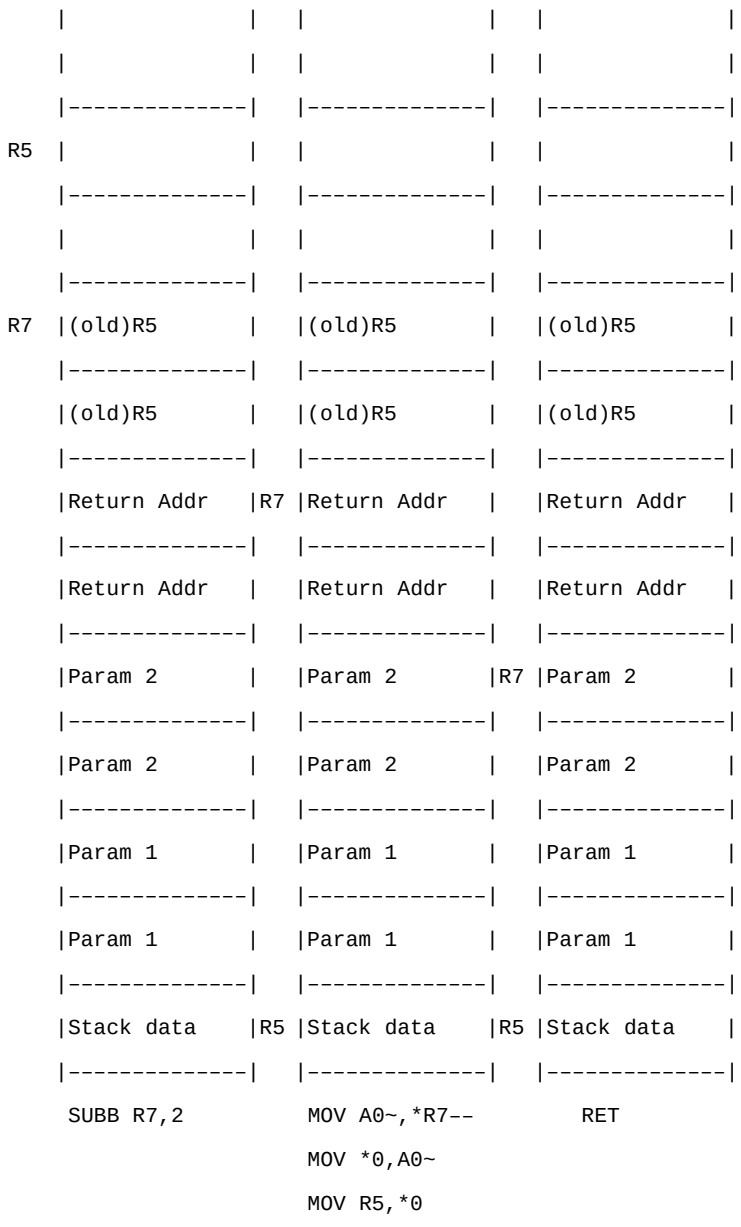
It is also important not to alter the contents of registers R5 and R7. R7 is the stack pointer and R5 is a frame pointer used in C to C function calls. Parameters are passed on the stack and the return value is always int and always located in a0. The stack usage for function calls is as follows.

C to C function call. The stack is shown after the operation on the bottom is performed.


R7,R5
Stack data
R5
Stack data
R5
Stack data
R7
Param 1
Param 1
Param 1
R7
Param 2
Param 2
Param 1
Before call
Parameter 1
Parameter 2

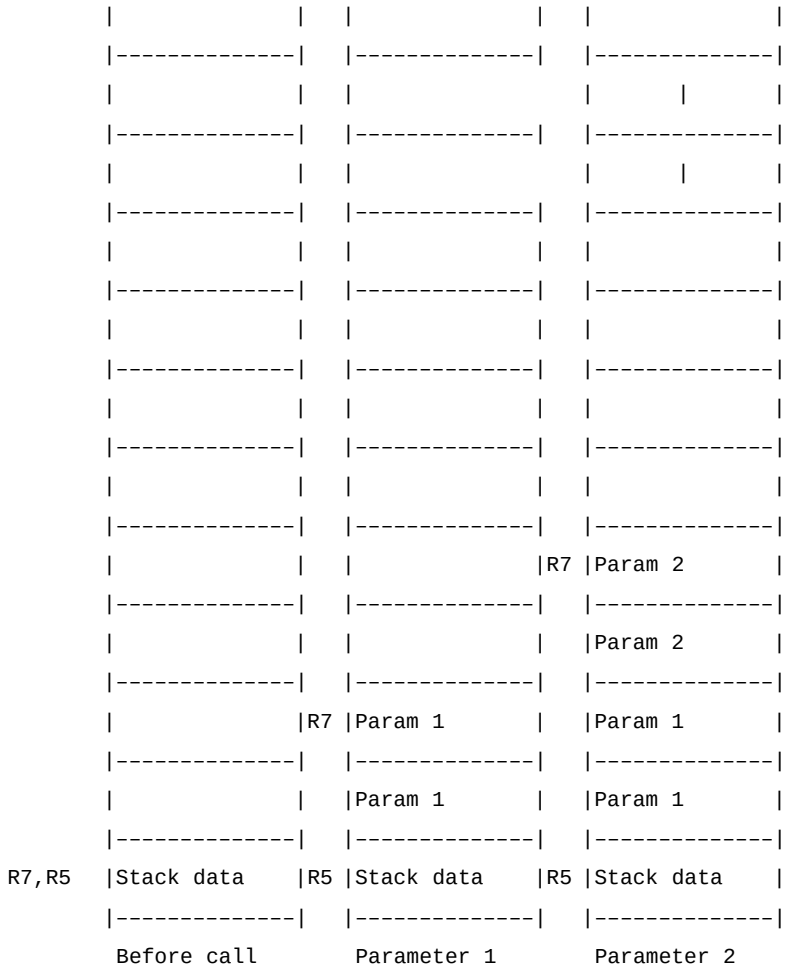

R7
R5
Return Addr
Param 2
Param 1
Stack data
R7
R5,R7
R5
(old)R5
<– This is the SP before the C function call.
Function call
ADDB R7,2
MOV *R7++,R5
MOV *0,R7
MOV R5,*0

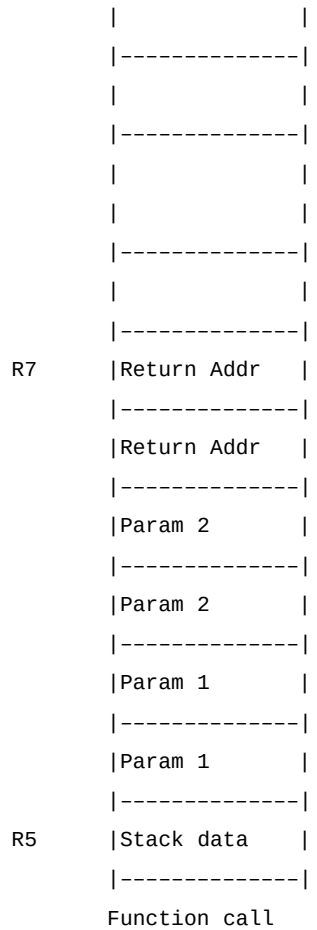
C to C function return (in ronco_return).



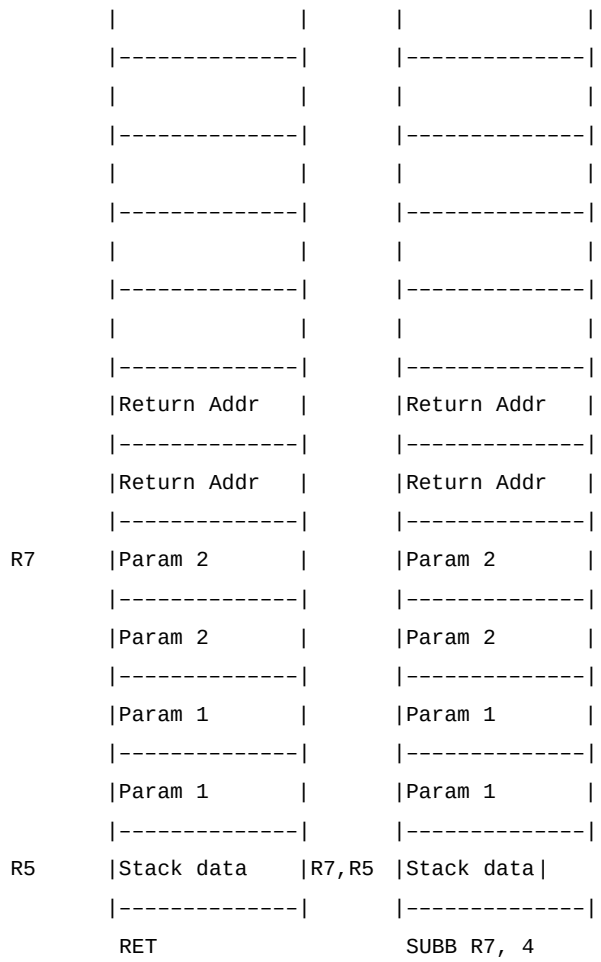
```
                |--------------|
                |              |
                |--------------|
                |              |
                |--------------|
                |              |
                |--------------|
                |(old)R5       |
                |--------------|
                |(old)R5       |
                |--------------|
                |Return Addr   |
                |--------------|
                |Return Addr   |
                |--------------|
                |Param 2       |
                |--------------|
                |Param 2       |
                |--------------|
                |Param 1       |
                |--------------|
                |Param 1       |
                |--------------|
R7,R5           |Stack data    |
                |--------------|
                 SUBB R7,4
```

C to ASM function call. The stack is shown after the operation on the bottom is performed.




R7
R5
Return Addr
Return Addr
Param 2
Param 2
Param 1
Param 1
Stack data
Function call

C to ASM function return



5.7 C–– Efficiency

C–– allows top-level control code to be written in a C-like language. It does have limitations though. Assembly routines are needed to support C function calls for operations such as reading/writing ports and speaking. The assembly code produced by the C–– compiler is not optimized and will not be as efficient as hand written assembly. Because C–– is not optimized, time critical operations should always be written in assembly.

Since the compiler does not optimize the assembly, writing C–– code in different ways results in different levels of overhead. While and do-while loops have similar overhead (one conditional jump and 1 unconditional jump per

repetition), but for loops are implemented with much greater overhead (one conditional jump and three unconditional jumps per repetition.) For this reason, it is best to replace for loops with while loops. *(This was not done in the example projects for the sake of readability and to provide an example of a C–– for loop.)* If the number of repetitions is both fixed and small, the code will execute faster if the loop is unwrapped. Switch statements and if-else blocks have similar overhead. Switch statements are slightly more efficient because the values being compared are only looked up once, while an if-else block looks up the values for each comparison. Switch statements do not use a table lookup; they use a fall through structure like an if-else block. Because of the fall through structure of switch and if-else blocks, items occurring first are executed with less overhead than items occurring last. If it is known that certain cases will occur more frequently than others, the code will execute fastest if the most frequently occurring cases are put before the less frequently occurring ones.

Space for global variables is allocated at compile time. Space for local variables is allocated on the stack at run time. This means that the compiler will not generate a warning if local variables exceed the available RAM. The compiler will generate an error message if the global variables exceed the available RAM. Caution must be used to avoid overflowing the stack by allocating too many local variables. During a call, parameters, return address, local variables, and the frame pointer are stored on the stack using a stack frame. The stack frame structure allows recursive calls, but the elegant solution provided by a recursive program is often offset by inefficiency. Using recursive calls is not recommended with the C–– compiler.

Dividing the program into too many functions can be inefficient also. It may be stylish to separate portions of the program into functions based on what they are designed to do, but unless the functions will be used in multiple places in the program, it is better not to make a function call. There is a tradeoff between ROM usage and RAM usage depending on the number of times a function will be needed. Using a function call requires more RAM and instruction overhead. Not using a function call can require more ROM depending on the size of the function and the number if times it is used.

5.7.1 Real Time Clock Example

The C–– clock works as follows. The Timer2 ISR is set to fire at 1-second intervals. Inside the ISR a counter is incremented by one each time it fires. An assembly routine in cmm1.asm (_getSecondsPassed) disables the interrupts, retrieves the counter, resets it, and turns the interrupts back on. The C–– program calls getSecondsPassed() whenever it is not busy and uses the return value to update the clock. This keeps the assembly code to a minimum and allows all of the calculations to be handled in C––. The interrupts are disabled when the counter is being read to prevent possible loss of time.

```
_getSecondsPassed
   rpt2-2; interrupt can still fire for 2 cycles
   intd  ; leaving these out can cause loss of a second
   mova0~, *seconds_passed
   zaca0
   mov*seconds_passed, a0
   inte
   mova0, a0~
   ret
```

The example is divided up into three projects. The first one is a minimal implementation. It does not have support for speech, LCD, key scanning, or setting the time. It offers minimum functionality to keep the number of files small. It is meant to show the basics of a C–– project.

The second project adds speech and key scanning. The speech provides output and the key scanning is used for input. Adding speech synthesis increases the number of files in the project dramatically, but the C–– is still similar. The main changes that relate to C–– are addition of a routine in cmm1.asm to read buttons on Port D and addition of a routine to speak from C––.

The third project adds LCD support. It offers the same speaking abilities as the third project, but uses an LCD screen for additional output. It also demonstrates the use of arrays in C––.

Example 5–1. First Project

The project is of limited use because there is no way to read the time or change the time without using a scanport. It does provide a good example of a C–– project that contains a few simple files.

A minimum implementation of the real time clock contains the following files.

- [Root]
  - cmm1.asm
  - cmm1_ram.irx
  - flags.irx
  - main.cmm
  - main.irx
  - main_ram.irx
  - mainasm.asm
  - vroncof2.asm
  - rtc.rpj
  - [modules]
    - [general]
      - init.asm
      - io_ports.irx
    - [isr]
      - tim2_isr.asm
- [ram]
  - ram.h
  - ram.irx

cmm1.asm	Assembly to support C–– function calls
cmm1_ram.asm	Allocates RAM for use in cmm1.asm
flags.asm	Flags used in init.asm and for speech routines
main.cmm	C–– program
main.irx	Mnemonics for switches and ports used in main.cmm
main_ram.irx	Allocates RAM for ISRs and mainasm.asm
vroncof2.asm	Assembly routines for built in C–– functions and ISR vector table
rtc.rpj	Project file generated by MSP50C6xx development tool
[modules]	Directory for ISRs, general purpose files, and plugable modules
[general]	Directory holding general-purpose files for initialization and mnemonics
init.asm	Initializes the clock on startup
io_ports.irx	Mnemonics for the io ports
[isr]	Directory for ISRs
tim2_isr.asm	Timer 2 interrupt service routine
[ram]	Directory for top level ram allocation files
ram.h	Holds the bogus array used by C––
ram.irx	Top level memory allocation

Seven of the files are important to the functionality of this project. The Timer2 ISR (tim2_isr.asm) forms the basis for the RTC so it will be discussed first.

```
timer2_isr
   mov *save_tim2_stat,STAT   ;save status
   mov    *save_tim2_a0,a0    ;save a0
   ; timer fired so 1 second passed
   ; update the variable storing the seconds passed so far
   mov a0, *seconds_passed
   add a0, a0, 1
   mov *seconds_passed, a0
   mov    a0,*save_tim2_a0    ;restore a0
   mov STAT,*save_tim2_stat   ;restore status
   inte               ;turn interrupts back on
iret
```

The Timer2 ISR is configured to fire at 1 second intervals. Each time the ISR executes, it saves any registers that it will modify, increments the RAM location seconds_passed, and restores the registers it modified.

The second important file is main_ram.irx. It is used to allocate RAM for seconds_passed and for saving and restoring registers in the Timer2 ISR.

```
;*************************************************************
; MAIN_RAM.IRX
;
; Start of memory for MAIN module is defined in
;      include   "..\ram\ram.irx"
;*************************************************************
; Timer 2 interrupt variables
save_tim2_stat      equ RAMSTART_CUSTOMER + 2 * 1
save_tim2_a0    equ save_tim2_stat + 2 * 1
seconds_passed      equ save_tim2_a0 + 2 * 1
RAMSTART_CMM1   equ seconds_passed
    include "cmm1_ram.irx"
; End of RAM
RAMEND_CUSTOMER     equ RAMEND_CMM1
RAMLENGTH_CUSTOMER  equ  RAMEND_CUSTOMER – RAMSTART_CUSTOMER
```

Any additional ram that is used in an ISR or in mainasm.asm should be allocated here. RAM is allocated by making a new label and setting it equal to the previous label plus an offset. A variable called some_variable could be allocated by changing

```
seconds_passed      equ save_tim2_a0 + 2 * 1
RAMSTART_CMM1   equ seconds_passed
```

to

```
seconds_passed      equ save_tim2_a0 + 2 * 1
some_variable   equ seconds_passed + 2 * 1
RAMSTART_CMM1   equ some_variable
```

The next important file is vroncof2.asm. Most of this file is used to support standard C functionality and will not need to be changed. The part that will

change is the table of interrupt vectors. At the top of the file is a list of interrupt labels. The ones that are not used are commented out with a semicolon.

```
;  external  DAC_ISR
;  external  timer1_isr
   external  timer2_isr
;  external  pd2
;  external  pd3
;  external  portF
;  external  pd4
;  external  pd5
```

At the bottom of the file are a dummy interrupt routine and the interrupt vector table.

```
pd2
pd3
portF
pd4
pd5
DAC_ISR
timer1_isr
;timer2_isr
   nop
   INTE
   iret
   AORG 07F00h
   DATA _EQ,_NE,_LT,_LE,_GE,_GT,_ULT,_ULE,_UGE,_UGT,_LNEG
   DATA _EQS,_NES,_LTS,_LES,_GES,_GTS,_ULTS,_ULES,_UGES,_UGTS
   DATA _DIV,_DIVU,_EXTB,_ASR
   AORG  07FF0h
   data  DAC_ISR   ; the DAC interrupt is used for synthesis
   data  timer1_isr ; this is the timer1 isr.
   data  timer2_isr   ; this is the timer2 isr.
   data  pd2
   data  pd3
   data  portF
   data  pd4
   data  pd5
   aorg  0x7ffe
   data  0x1ffff         ;ROM protection word (0x7ffe)
   data  init614         ;reset address (0x7fff)
```

Notice that timer2 was not commented out at the top of the file but it is commented out in the dummy interrupt routine. External interrupt routines are switched on by not commenting them at the external statement at the top of the file and commenting them in the dummy interrupt routine. They are switched off by commenting their external statement at the top and not commenting the label in the dummy routine at the bottom. Note that this does not enable or disable an interrupt, it just controls what is executed when it fires. Interrupt routines will be enabled and disabled in the next file, but it is important to provide a dummy routine for unused interrupts in case a programming error causes them to be accidentally enabled.

Mainasm.asm contains the most complex assembly. It is responsible for initializing assembly variables, enabling or disabling interrupts, and setting up any timers or I/O ports. It also enables the interrupts. The part that is important to the project, _goasm, is called at the beginning of the C–– main routine.

```
;*************************************************************
;
; Main program
;
; Set i/o for any peripherals (eg ADC chip, flash card or LCD)
; and initialize variables as necessary. All user code should
; start here.
;*************************************************************
;
; clear the seconds passed counter
   zac a0
   mov *seconds_passed,a0
; Set TIMER2 to run from the RTO/CTO (32 kHz) and with a 1000ms period. Set this
by
; loading TIM2 with (32768 x 1000/1000), minus 1.
   in  a0,IntGenCtrl
   or  a0,TIM2REFOSC       ;set bit 9, CTO clock (32 kHz)
   and a0,~TIM2ENABLE         ;clear bit 11, TIM2 enable
   out IntGenCtrl,a0
   mov a0,32768 – 1        ;setup a 250ms period
   out TIM2,a0          ;load TIM2 and PRD2 in one step
   in  a0,IntGenCtrl
or  a0,TIM2IMR + TIM2ENABLE    ;set bit 2 (TIM2 interrupt
    ;enable) and bit 11
   out IntGenCtrl,a0
   inte
   ret
```

In this example, it clears seconds_passed, which was used in the first file (timer2_isr.asm), sets up timer2 to run at a 1Hz interval, and enables the interrupts.

The fifth important file is cmm1.asm. This file is responsible for supporting C–– to assembly function calls. It takes parameters passed on the stack, processes them, and returns a 16-bit value in A0. In C–– the 16-bit return value is always of type int.

```
;*************************************************************
;
; CMM1.ASM
;
; Revision 1.00
;*************************************************************
;
   rorg   0x0
   global    _getSecondsPassed
   include      ”ram\ram.irx”
; retrieve the seconds that have passed, and reset the counter
_getSecondsPassed
   rpt2–2 ; interrupt can still fire for 2 cycles
   intd   ; leaving these out can cause loss of a second
   mov a0~, *seconds_passed
   zac a0
```

```
    mov *seconds_passed, a0
    inte
    mov a0, a0~
    ret
```

The file only has one C–– callable function, getSecondsPassed. The function reads the value in seconds_passed and returns it in A0. All C–– functions have an underscore preceding their name in assembly. The underscore is ignored when programming in C––. In C–– a call to this function would look like

```
int result = getSecondsPassed();
```

Notice that the underscore is not used here because C–– is being used instead of assembly. getSecondsPassed() has very simple functionality, but it illustrates several important points. First, interrupts are disabled with the intd instruction. This is extremely important because it is not possible to read the value in seconds_passed and clear it in an atomic operation. If the value is read and the timer fires before it is cleared, one second will be lost. The next important feature to note is the inclusion of rpt2–2 before the intd instruction. Because of pipeline latency, interrupts can still fire for two clock cycles after an intd instruction. The rpt temporarily disables interrupts and ensures that an interrupt does not fire and execute an inte before the intd makes it through the pipeline. Disabling interrupts ensures that the timer will not fire while the value in seconds_passed is being read and altered.

The sixth file, cmm1_ram.asm, allocates memory for cmm1.asm.

```
;*************************************************************
; CMM1_RAM.IRX
;
; Start of memory for asmroutines module is defined in
;      include   "..\ram\ram.irx"
;*************************************************************
;
; Variables
; End of memory
RAMEND_CMM1   equ RAMSTART_CMM1
RAMLENGTH_CMM1   equ RAMEND_CMM1 – RAMSTART_CMM1
```

In this project, cmm1.asm did not use any RAM, but it can be allocated just like the RAM for the ISRs. For example, a variable named tempa could be allocated as follows.

```
; Variables
tempa     equ RAMSTART_CMM1 + 2 * 1
; End of memory
RAMEND_CMM1   equ tempa
```

The last file is the C–– program, main.cmm. This provides all of the top level functionality for the project. Once all of the previous supporting files have been written, writing the C–– program is very much like writing a regular C program.

```
/*************************************************************
```

```
; MAIN.CMM
; Revision 1.00
*************************************************************/
#include  ”ram\ram.h”
cmm_func  goasm(); // an pseudo main asm routine
cmm_func  getSecondsPassed(); // Retrieves the counter maintained
                // by the Timer2 ISR and resets the
                // counter.
int days=0;
int hours=12;
int minutes=0;
int    seconds=0;
int ampm=0;
/*******************************************
/ Updates time variables for clock ticks that
/ have occured.
*******************************************/
cmm_func updateTime(){
   seconds=seconds+getSecondsPassed();
   while(seconds>59){
      seconds=seconds-60;
      minutes++;
      if(minutes>59){
         minutes=0;
         hours++;
         if (hours == 12){
            if(ampm==0){
               ampm=1;
            }
            else{
               ampm=0;
               days++;
            if(days>6){
               days=0;
            }//end days
            }
         }
         if (hours>12){
            hours=1;
         }//end hours
      }//end minutes
   }//end seconds
}
cmm_func main()
{
   goasm(); // run any assembly stuff that needs to be run
   while(1){ // infinite loop
      updateTime();
   }
}
```

The include statement at the top of the program is for memory allocation purposes. The C–– compiler is not aware that RAM has been allocated for assembly and must be kept from overwriting it. This is done with an integer array called bogus. The array is set to the size of the RAM allocated for

assembly divided by two because C–– integers are 16 bit. The perl script in the main project directory can be used to resize bogus automatically or it can be done manually. To use the perl script, build the project after making any changes to assembly ram allocation. Run the perl script and then rebuild the project. To manually adjust bogus, build the project and then examine the list file mainasm.lst. Find RAMEND_ASM in the cross reference table and use it to replace the value in the define statement in ram.h. Rebuild the project to put the changes into effect. This only needs to be done when changes are made to assembly RAM allocation. Changes to C–– or assembly code other than RAM allocation do not require adjustments to bogus.

The next items in the program are function prototypes. All C–– functions have a return type of int (16 bit) and are declared with the mnemonic cmm_func. The first one is goasm(). Notice that there is no leading underscore because it is being called from C–– instead of assembly. The second one is the function in cmm1.asm for reading the value of seconds_passed.

Global variables are defined next. An integer is used to keep track of each element of the time. Global variables can be initialized when they are declared.

The function updateTime() is used to update the time. It calls getSecondsPassed() to determine the number of seconds that have passed since the time has been updated. It then recalculates the time variables (hours, minutes, etc.) updateTime() does not pass any values when it returns although it is technically of type int, like all C–– functions.

The main() function is the starting point for user code. After the 6xx part has been initialized main() is called from vroncof2.asm. The first call from main() is to doasm() which is in mainasm.asm. This is the function that sets up the timer and initializes seconds_passed. The program then goes into an infinite loop where updateTime() is called. In later projects, this infinite loop will be expanded to scan keys and write to the LCD.

Example 5–2. Second Project (C— With Speech)

Adding speech to the first project increases functionality, but also increases the complexity of the project.

- [Root]
  - cmm1.asm
  - cmm1_ram.irx
  - flags.irx
  - main.cmm
  - main.irx
  - main_ram.irx
  - mainasm.asm
  - vroncof2.asm
  - rtc.rpj
  - [dsp]
    - [celp]
      - celp.irx
      - celp4.obj
    - [common]
      - util.obj
      - util2.obj
    - [general]
      - dsp_var.irx
      - dsputil.asm
      - getbits.asm
      - speak.asm
      - speak.irx
      - spk_ram.irx
    - [melp]
      - melp.irx
      - melp.obj
  - [modules]
    - [general]
      - init.asm
      - io_ports.irx
      - sleep.asm
    - [isr]
      - tim2_isr.asm
      - dac_isr.asm
      - tim1_isr.asm
  - [speech]
    - [celp]
      - ampm.qfm
      - days.qfm
      - ones.qfm
      - teens.qfm

tens.qfm
- [melp]
  - ampm.qfm
  - days.qfm
  - ones.qfm
  - teens.qfm
  - tens.qfm
- [ram]
  - ram.h
  - ram.irx

Descriptions of files that are also in Project 1 have been omitted.

[dsp] Directory holding files for speech synthesis.

[celp]	Directory holding files for celp synthesis.
celp.irx	Mnemonics used by celp.obj.
celp4.obj	Celp synthesis routines.
[common]	Directory holding utility routines.
util.obj	Utilities used for synthesis.
util2.obj	Utilities used for synthesis.
[general]	Directory for non-coder-specific routines.
dsp_var.irx	Constants used by the synthesis routines.
dsputil.asm	Routines common to the synthesis algorithms.
getbits.asm	Routines for requesting speech data.
speak.asm	Routines for speaking a phrase.
speak.irx	Combines irx files for each synthesis algorithm.
speak_ram.irx	Allocates RAM for speech synthesis.
[melp]	Directory holding files for melp synthesis.
melp.irx	Mnemonics used by melp.obj.
melp.obj	Melp synthesis routines.
sleep.asm	Functions to enter sleep modes.
dac_isr.asm	DAC interrupt service routine.
tim1_isr.asm	Timer 1 interrupt service routine.
[speech]	Directory holding speech data.
[celp]	Directory holding CELP speech data.
[melp]	Directory holding MELP speech data.
ampm.qfm	Speech file of AM and PM .
days.qfm	Speech file of Sun–Sat.
ones.qfm	Speech file of 0–9.
teens.qfm	Speech file of 10–19.
tens.qfm	Speech file of 20, 30, 40, 50.

Five of the important files from the first project have been modified and there are many new files.

In main_ram.irx, two variables were added to save and restore r3 and r5 when speaking. These registers are used by C–– so it is a good idea to save and restore them in case they are modified by the speech routines. This is a good example of adding RAM for use by cmm1.asm.

```
;************************************************************
; MAIN_RAM.IRX
;
; Start of memory for MAIN module is defined in
;      include   "..\ram\ram.irx"
;************************************************************
; Timer 2 interrupt variables
save_tim2_stat      equ RAMSTART_CUSTOMER + 2 * 1
save_tim2_a0     equ save_tim2_stat + 2 * 1
seconds_passed   equ save_tim2_a0 + 2 * 1
csave_r3      equ seconds_passed + 2 * 1
csave_r5      equ csave_r3 + 2 * 1
RAMSTART_CMM1    equ csave_r5
    include "cmm1_ram.irx"
; End of RAM
RAMEND_CUSTOMER  equ RAMEND_CMM1
RAMLENGTH_CUSTOMER  equ    RAMEND_CUSTOMER – RAMSTART_CUSTOMER
```

The new variables, csave_r3 and csave_r5 were added by using the mnemonic for the previous variable plus an offset.

The next modified file is vroncof2.asm. Here new interrupt service routines were added. This project adds speech so the DAC_ISR needs to be added. Timer 1 is also used for waking up from sleep routines so it was also added. At the top of the file their labels were uncommented.

```
   external  DAC_ISR
   external  timer1_isr
   external  timer2_isr
;  external  pd2
;  external  pd3
;  external  portF
;  external  pd4
;  external  pd5
   external  init614
   external  _main0
```

At the bottom of the file, their labels were commented out of the dummy interrupt routine.

```
pd2
pd3
portF
pd4
pd5
;DAC_ISR
;timer1_isr
;timer2_isr
   nop
```

```
    inte
    iret
```

Cmm1.asm was modified to include routines for sleeping and speaking from C––.

```
    global    _inportD
    global    _getSecondsPassed
    global    _sleepQuarterSec
    global    _speakDays
    global    _speakOnes
    global    _speakTens
    global    _speakTeens
    global    _speakAMPM
```

New C–– callable functions were declared global.

```
    external  sleep_light
    external  speak
```

Assembly routines that will be called are declared external.

```
    include      "speech\celp\days.qfm"
    include      "speech\celp\ones.qfm"
    include      "speech\celp\teens.qfm"
    include      "speech\celp\tens.qfm"
    include      "speech\celp\ampm.qfm"
```

Include statements were used to add speech files for all of the phrases that the clock will need to say.

```
_sleepQuarterSec
    mov a0,8192 – 1      ;setup a 250ms period
    out TIM1,a0         ;load TIM1 and PRD1
    mov a0, TIM1IMR
    call   sleep_light
    nop
    ret
```

A routine was added to sleep for a quarter second using Timer 1 to wake up. The program loads the period into the timer 1 period register, sets the wake-up mask in a0, and calls sleep_light, which is in sleep.asm.

```
_speakDays
    ; back up important registers
    mov *csave_r5, r5    ; protect r5
    mov *csave_r3, r3    ; protect r3
   mov a0, *r7 – 2 ; synthesis table offset
    add    a0, _days_table
    mov a0, *a0
    ZAC A0~
    CALL SPEAK
    ; restore important registers
    mov r3, *csave_r3    ; restore r3
    mov r5, *csave_r5    ; restore r5
    ret
_days_table ; table for table lookup
```

```
   DATA   MON ;0
   DATA   TUE ;1
   DATA   WED ;2
   DATA   THU ;3
   DATA   FRI ;4
   DATA   SAT ;5
   DATA   SUN ;6
```

C–– callable speech routines, like the above for speaking days were added. An integer phrase number is passed on the stack. The routines get this value from the stack and do a table lookup to get the address of the correct phrase. The address is loaded into a0 and a0~ is cleared to indicate that the speech data is in internal ROM. Then speak, which is located in speak.asm, is called.

The final assembly file that was modified was mainasm.asm. The only change was setting up Timer 1 and enabling the Timer 1 interrupt. The configuration of Timer 1 is similar to the configuration of Timer 2.

The high level program, main.cmm, was then modified to utilize the new functionality.

```
cmm_func main()
{
   goasm(); // run any assembly stuff that needs to be run
   while(1){ // infinite loop
      if(!(inportD()&SW1)){
         setTime();
      }
      if(!(inportD()&SW2)){
         speakTime();
      }
      updateTime();
   }
}
```

The main() routine now reads keys by calling the inportD() which was added to cmm1.asm as _inportD. The value is compared against a constant to see if a certain key was pressed and then the function for that key is called. Key checking and updates to time are all done inside the infinite loop.

Speaking the time is very simple using the routines that were added to cmm1.asm.

```
cmm_func speakHours(){
       if( hours<10){ // 1–9
              speakOnes(hours);
       }
       else{ // must be 10, 11, or 12
              speakTeens(hours–10);
       }
}
```

The appropriate speak function is called and the parameters are passed to it. The program flow does not return to C–– until the speech file has finished

playing. In some cases speech files can be played to debounce keys. This is why there is no delay in the main() function. Pressing SW2 calls a function, but the switch will not be read again until the time has been spoken so there is no need for a delay there.

Example 5–3. Third Project (C— with an LCD)

The main difference between this project and the second project is the addition of an LCD display. The variables storing the time were also changed to an array of ints instead of separate int variables to demonstrate the use of C–– arrays. This is not the clearest or easiest way to keep track of the time. It was added as an example of C–– arrays. Multidimensional arrays are not supported in C––, but the same functionality can be achieved by multiplying and adding the indices. For example, if an array is defined as:

```
int a [3*4]; // equivalent to int a [3][4] in C
```

Then the element at row x column y can be accessed by using index = rowNum * row + column.

```
value = a[3*1+0]; // equivalent to value = a[1][0] in C
```

- [Root]
  - cmm1.asm
  - cmm1_ram.irx
  - flags.irx
  - main.cmm
  - main.irx
  - main_ram.irx
  - mainasm.asm
  - vroncof2.asm
  - rtc.rpj
  - [dsp]
    - [celp]
      - celp.irx
      - celp4.obj
    - [common]
      - util.obj
      - util2.obj
    - [general]
      - dsp_var.irx
      - dsputil.asm
      - getbits.asm
      - speak.asm
      - speak.irx
      - spk_ram.irx
    - [melp]

- melp.irx
- melp.obj
- [modules]
  - [general]
    - init.asm
    - io_ports.irx
    - sleep.asm
  - [isr]
    - tim2_isr.asm
    - dac_isr.asm
    - tim1_isr.asm
  - [lcd]
    - lcd.asm
    - lcd.irx
    - lcd_ram.irx
- [speech]
  - [celp]
    - ampm.qfm
    - days.qfm
    - ones.qfm
    - teens.qfm
    - tens.qfm
  - [melp]
    - ampm.qfm
    - days.qfm
    - ones.qfm
    - teens.qfm
    - tens.qfm
- [ram]
  - ram.h
  - ram.irx

Descriptions of files that are also in Project 2 have been omitted.

[lcd]	Directory holding files for writing to an LCD screen.
lcd.asm	Routines for writing to an LCD screen.
lcd.irx	Mnemonics used by lcd.asm.
lcd_ram.irx	Allocates RAM for lcd.asm.

The only changes to the assembly are in mainasm.asm and in cmm1.asm. In mainasm.asm, two calls are made to setup and initialize the lcd. To allow this, the labels for the routines were declared external.

```
   external  lcd_setio
   external  lcd_init
```

In _goasm, they are then called to initialize the LCD before it is used.

```
; set up the LCD
   call   lcd_setio
   call   lcd_init
```

In cmm1.asm, simple routines for writing characters and numbers were added, along with routines to bring the cursor to the beginning of the first and second row.

```
_writeNum
   mov a0, *r7 – 2
   call   lcd_wrbcd2
   ret
_writeCharacter
   mov a0, *r7 – 2
   call   lcdwchr
   ret
_rowZero
   call   lcd_row0
   ret
_rowOne
   call   lcd_row1
   ret
```

_writeNum and _writeCharacter get a value to write from the stack and then call routines in lcd.asm. _rowZero and _rowOne simply call routines in lcd.asm.

main.cmm has been modified by the addition of a function, showTime().

```
/*********************************************
/ Display the time on the LCD
*********************************************/
cmm_func showTime(){
   int temp;
   rowZero();
   writeNum(time[WIDTH*0+0]); //hours
   writeCharacter(':');
   writeNum(time[WIDTH*0+1]); //minutes
```

```
writeCharacter(':');
writeNum(time[WIDTH*0+2]); //seconds
writeCharacter(' ');
if(time[WIDTH*1+1]==0){ //ampm
   writeCharacter('A');
}
else{
   writeCharacter('P');
}
writeCharacter('M');
writeCharacter(' ');
switch(time[WIDTH*1+0]){ //days
          case 0:
      writeCharacter('M');
      writeCharacter('O');
      writeCharacter('N');
      break;
   case 1:
      writeCharacter('T');
      writeCharacter('U');
      writeCharacter('E');
      break;
   case 2:
      writeCharacter('W');
      writeCharacter('E');
      writeCharacter('D');
      break;
   case 3:
      writeCharacter('T');
      writeCharacter('H');
      writeCharacter('U');
      break;
   case 4:
      writeCharacter('F');
      writeCharacter('R');
      writeCharacter('I');
      break;
   case 5:
      writeCharacter('S');
      writeCharacter('A');
      writeCharacter('T');
      break;
   case 6:
      writeCharacter('S');
      writeCharacter('U');
      writeCharacter('N');
      break;
}

switch(pendulum){
   case 0:
      writeCharacter(' ');
      writeCharacter('|');
      rowOne();
      for(temp=0; temp<16; temp++) writeCharacter(' ');
      writeCharacter('o');
```

```
            writeCharacter(' ');
            break;
         case 1:
            writeCharacter(' ');
            writeCharacter('(');
            rowOne();
            for(temp=0; temp<17; temp++) writeCharacter(' ');
            writeCharacter('o');
            break;
         case 2:
            writeCharacter(' ');
            writeCharacter('|');
            rowOne();
            for(temp=0; temp<16; temp++) writeCharacter(' ');
            writeCharacter('o');
            writeCharacter(' ');
            break;
         case 3:
            writeCharacter(' ');
            writeCharacter(')');
            rowOne();
            for(temp=0; temp<15; temp++) writeCharacter(' ');
            writeCharacter('o');
            writeCharacter(' ');
            writeCharacter(' ');
            break;
      }
      rowOne();
}
```

setTime() was also modified to place an indicator on the LCD below the value that is being set. updateTime() was modified to call showTime() after the time is updated.

5.8 Beware of Stack Corruption

MSP50C614/MSP50P614 stack (pointed by **R7** register) can easily get corrupted if care is not taken. Notice the following table read code:

```
SUBB R7, 4
MOV A0, *R7––
ADD A0, address
MOV A0, *A0
ADD A0, *R7––
MOV A0, *A0
RET
```

This code will work perfectly well if no interrupts happen before **SUBB** and **MOV** instruction. If interrupts do happen between **SUBB** and **MOV** instructions, the parameter in the stack is corrupted by the return address pushed by the hardware. This problem may not be easily observed in the system level. But once it happens, it is very difficult to debug. Use the following method to modify stack pointer instead:

```
MOV A0, *R7 + –2 * 2
ADD A0, address
MOV A0, *A0
ADD A0, *R7 + –2 * 1
MOV A0, *A0
RET
```

This method will not have the stack corruption problem since the **MOV** instruction performs the entire operation either before or after an interrupt.

5.9 Reported Bugs With Code Development Tool

The following are reported bugs for code development tool version 2.39.

Breakpoint: Placement of hardware breakpoints is important for reliable operation. Pipeline latency and sleep modes affect the scan logic and prevent hardware breakpoints from working in the following cases. Placing a breakpoint within two cycles of an IDLE instruction causes a breakpoint while the part is still in a low power mode. This will cause the code development tool to lose sync with the hardware. This is the same effect as trying to stop execution from the tool while the part is in a low power mode. Hardware breakpoints should not be placed within two cycles of a label accessed with a CALL instruction or as an ISR. This results in unreliable performance of the breakpoint. The breakpoint may not be triggered even though the code is executed. Placing the breakpoint a few lines into the routine solves this issue. Placing a hardware breakpoint within two cycles of a RET can be unreliable also. In general it is best not to place hardware breakpoints at the very beginning or end of subroutines or ISRs.

Hardware Presence: If the tool tries to communicate with the hardware and the hardware is not connected or is powered down it will lose sync. It is important to always keep the chip in the socket and powered unless the tool is stopped. The tool also communicates with the hardware after linking and when the tool is started.

Chapter 6

Applications

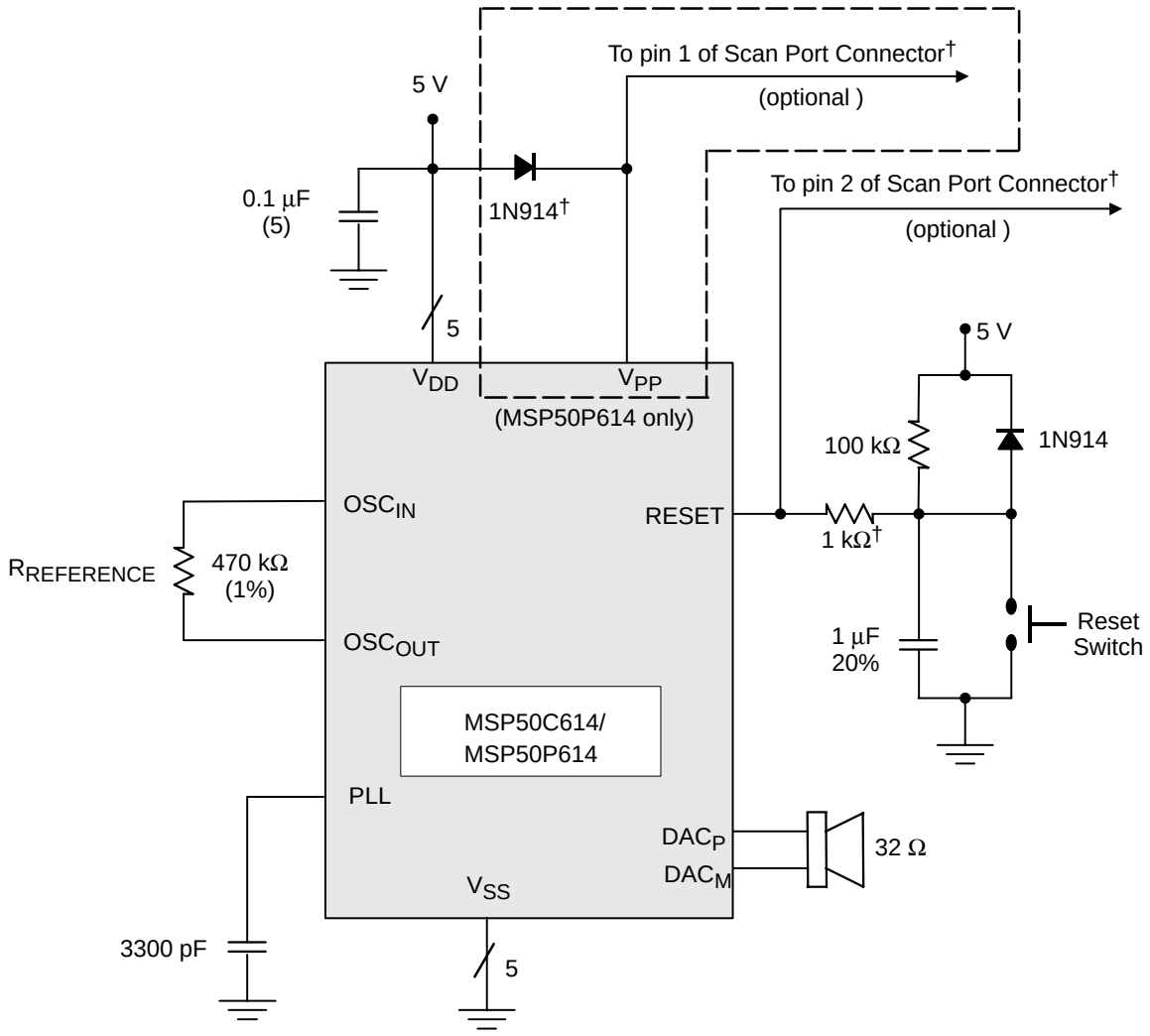
This chapter contains application information on application circuits, processor initialization sequence, resistor trim setting, synthesis code, memory overlays, and ROM usage.

Topic **Page**

6.1 Application Circuits 6–2
6.2 Initializing the MSP50C6xx 6–4
6.3 TI-TALKS Example Code 6–8
6.4 RAM Overlay 6–9

6.1 Application Circuits

Figure 6–1. Minimum Circuit Configuration for the C614/P614 Using a Resistor-Trimmed Oscillator



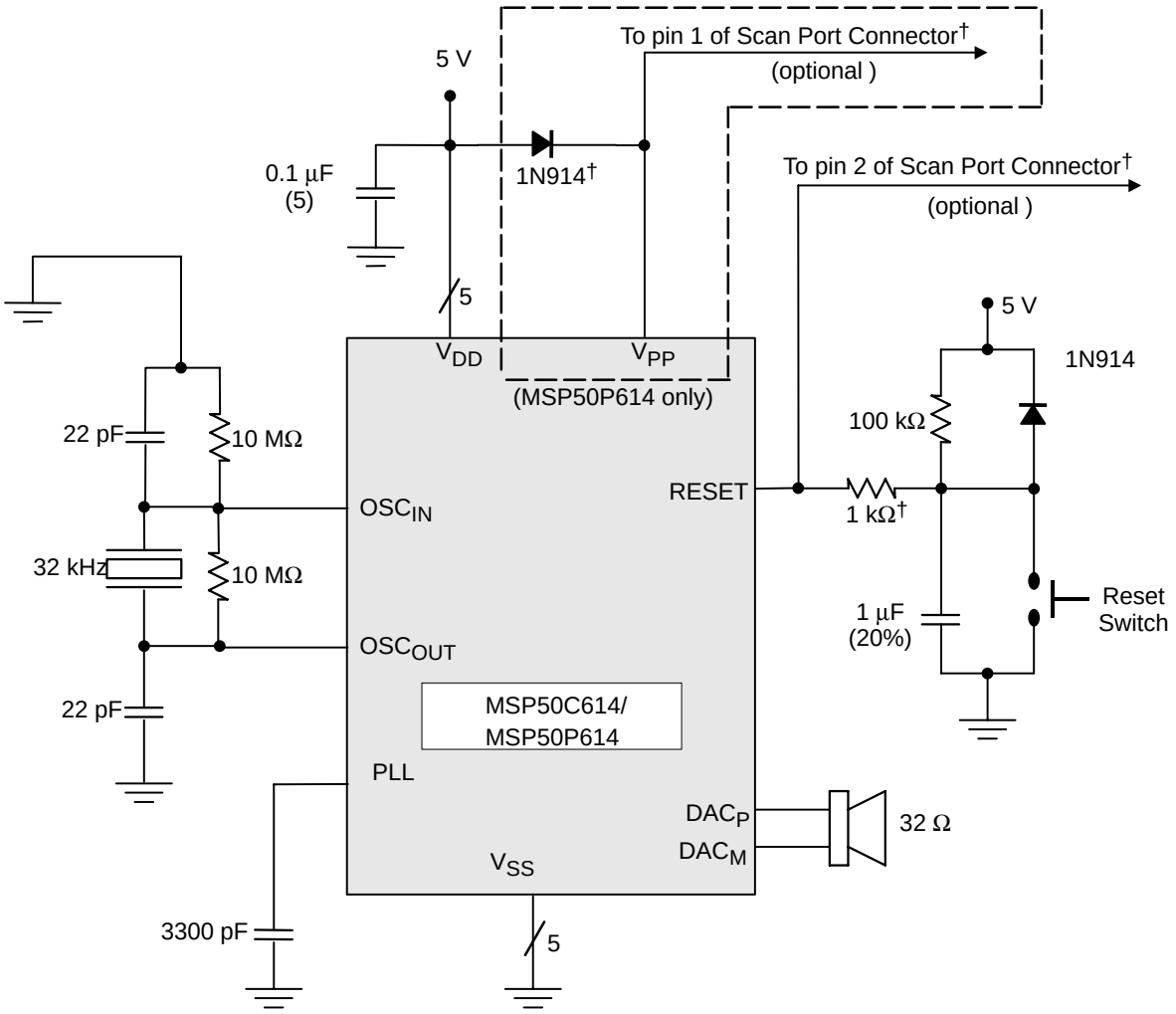
† The diode across V_{DD} and V_{PP} may be omitted (shorted), if the application does not require use of the scan port interface. The same applies for the 1-kΩ resistor which appears at the RESET pin; the resistor may be shorted if not using the scan port. However, the *footprint* for the resistor is *strongly* recommended for any MSP50C614 production board. Refer to the *Important Note regarding Scan Port Bond Out* appearing in Chapter 7.

Note, that there are five V_{DD} pins and five V_{SS} pins. Each of these should be connected, with the separate decoupling capacitors (0.1 µF) included for each V_{DD}.

It is of particular importance to provide a separate decoupling capacitor for the V_{DD}, V_{SS} pair which services the DAC. These pins are pad numbers 21 and 19, respectively. The relatively high current demands of the digital-to-analog circuitry make this a requirement.

An alternate circuit, for better clock-precision and better battery life, includes a crystal oscillator. See Figure 6–2.

Figure 6–2. Minimum Circuit Configuration for the C614/P614 Using a Crystal-Referenced Oscillator



† The diode across V_{DD} and V_{PP} may be omitted (shorted), if the application does not require use of the scan port interface. The same applies for the 1-kΩ resistor which appears at the RESET pin; the resistor may be shorted if not using the scan port. However, the *footprint* for the resistor is *strongly* recommended for any MSP50C614 production board. Refer to the *Important Note regarding Scan Port Bond Out* appearing in Chapter 7.

In any MSP50C614 application, it is important for certain components to be located as close as possible to the MSP50C614 die or package. These include any of the decoupling capacitors at V_{DD} (0.1 μF). It also includes all of the components in the crystal-reference network between OSC_{IN} and OSC_{OUT} (22 pF, 10 MΩ, 32 kHz).

6.2 Initializing the MSP50C6xx

The initialization code for the MSP50C6xx is in the file INIT.ASM, in the MODULES\GENERAL directory of the TI-TALKS code (see the following information).

The initialization routine does the following:

- Clears the status registers
- Clears all 32 accumulators
- Clears all 640 words of RAM
- Clears all system registers
- Sets the clock to run at 8.192 MHz. If CRO_FLAG is 1, the crystal oscillator is used. Otherwise, If CRO_FLAG is 0, the resistor-trimmed oscillator is used.
- Enables port F pullups
- Sets the DAC to 10 bits and turns it on
- Jumps to the label_main in MAIN.ASM

Note:

Care must be taken when branching to the init code to perform a software reset on parts using resistor trim. The resistor trim is set based on the value of fuses blown by the tester when the parts are manufactured. The P part does not have these fuses so initially the value at that location is zero. If the init routine encounters a zero it knows that it is running on a P part and sets the resistor trim to a constant value, RESISTORTRIM. This will always work properly after a hardware reset because all IO port locations are set to zero. If the programmer branches to the init code to perform a software reset, the value at 0x2F may not necessarily be zero. The IO addresses are not fully decoded on the P part, so writing to 0x2C (port G) also writes to 0x2D, 0x2E, and 0x2F. This means that the value may not be zero during a software reset. If this occurs, the init code will misidentify the P part as a C part and will use the value at 0x2C as the trim. This may cause the P part to run at the wrong speed. It is important to consider this if the init code is used as a software reset. The C part has fuses at location 0x2C and fully decoded IO port addresses so this problem will not occur on masked parts.

6.2.1 File `init.asm`

```
;*****************************************************************
; INIT.ASM
;
; Revision 1.04
;
; Modified from revision 1.03: if not CRO, we check port 0x2F
;                              to distinguish between P and
;                              C parts.
;
; Turn off TIMER 2 rather than leave it running.
;
; Modified to cope with 6 bit trim value.  Top 5 bits go to bits
; 15–11 in ClkSpdCtrl, LSB of trim goes to bit 9 in ClkSpdCtrl.
;
; A fairly basic but compact initialization routine for the 614.
; This sets the 614 to run at 8 MHz, 10 bit DAC at 8 kHz.
;
; Geoff Martindale, BP
; May 2000
;*****************************************************************
;!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
;!!!!!!!!!!!!!!!!!!!!!!    WARNING    !!!!!!!!!!!!!!!!!!!!!!!!!!!!
;!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
;! FOR RESISTOR TRIM USERS:
;! DO NOT WRITE TO PORT G PRIOR TO READING THE RTRIM VALUE!
;! THIS PRESERVES THE ZERO VALUE AT PORT 0x2F WHEN READING THE
;! TRIM VALUE (should be zero if P part, should be non–zero if
;! C part).
;!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
;!!!!!!!!!!!!!!!!!!!!!!    WARNING    !!!!!!!!!!!!!!!!!!!!!!!!!!!!
;!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
;*****************************************************************
; Start off by clearing all the RAM (and tags) and then zero
; every register.  The status register (STAT) must be cleared
; immediately upon power up.
;*****************************************************************
init614
   zac a0           ;clear a0
   mov *0x001,a0       ;clear second RAM location, leave first for C––
   mov STAT,*0x001        ;clear status register

   mov STR,32–2        ;set string register to loop 32 times
   zacs   a0           ;clear all accumulators

   out IFR,a0          ;clear pending interrupts
   out IntGenCtrl,a0      ;clear all interrupt mask bits, disable timers

   mov r0,0x000        ;point to beginning of RAM
   mov r4,RAM_SIZE – 2       ;do a loop RAM_SIZE times

   BEGLOOP
   rtag   *r0          ;reset tag
   mov *r0++,a0        ;clear the RAM
   ENDLOOP

   mov STR,0           ;clear string register
```

```
   mov ap0,0           ;clear accum pointer 0
   mov ap1,0           ;clear accum pointer 1
   mov ap2,0           ;clear accum pointer 2
   mov ap3,0           ;clear accum pointer 3

   mov r0,0            ;clear register 0
   mov r1,0            ;clear register 1
   mov r2,0            ;clear register 2
   mov r3,0            ;clear register 3
   mov r4,0            ;clear register 4
   mov r5,0            ;clear register 5
   mov r6,0            ;clear register 6
   mov r7,0            ;clear register 7

   mov sv,0            ;clear shift value register

   mov TOS,*0x000         ;clear top of stack register
   mov PH,*0x000       ;clear product high register
   mov MR,*0x000       ;clear multiplier register
;*************************************************************
; Choose the source for the reference oscillator.  Set the PLLM
; register accordingly (in this case for a CPU clock of 8 MHz)
; and then set TIMER 2 to a 200 ms period.
; Go to sleep (do an IDLE) and wake up when the clock has
; reached full speed and is stable.
;*************************************************************
#if CRO_FLAG
   mov a0,CROENABLE       ;enable crystal oscillator
#else                                          ;Use BIST to determine P or C part
  ;----- BOB 5/00 -----
        IN      A0,0x2F                        ;On uninitialized P parts, port
0x2F is zero
   andb   a0,0xff             ;only want lower 8 bits
        JNZ     ITS_A_C_PART

ITS_A_P_PART
   mov A0,RESISTORTRIM        ;for P614 the user supplies the trim value
        jmp     setup_trim                     ;Now set up the trim in ClkSpdCtrl

ITS_A_C_PART
        in      A0,RTRIM                       ;for C614 read trim value from
register
  ;----- BOB 5/00 -----

setup_trim
   and a0,0x3f            ;only want lower 6 bits
   mov a0~,a0          ;save a copy for later
   mov sv,10           ;need to shift left by 10
   shltpl a0,a0           ;bit 1 is now bit 11, bit 0 now bit 10
   or  a0,RTOENABLE       ;enable resistor-trimmed oscillator
   and a0,~IDLEBIT        ;clear bit 10

; GJM  1.10.99
; 6 bit trim resides in bits 15-11 and bit 9 (LSB of trim value)
   and a0~,a0~,0x01       ;look at bit 0 of trim value
   jz  trimbit0        ;do nothing if it is zero
   or  a0,0x0200       ;else set bit 9
trimbit0
#endif
```

```
   orb a0,0x7c             ;set PLLM for CPU clock of 8 MHz

   mov *save_clkspdctrl,a0    ;save the ClkSpdCtrl value for later, when
                  ;waking up from mid or deep sleep

   mov a0~,TIM2REFOSC + TIM2IMR   ;disable TIMER 2
   out IntGenCtrl,a0~
   mov a0~,6553         ;setup a 200 ms period
   out TIM2,a0~         ;load TIM2 and PRD2 in one fell swoop
   mov a0~,TIM2ENABLE + TIM2REFOSC + TIM2IMR
   out IntGenCtrl,a0~         ;use 32 kHz crystal as source, wake up from TIM2

   out ClkSpdCtrl,a0       ;set clock to full speed!

   idle                ;go to sleep...

   nop              ;wake up 200 ms later, clock running at full speed
   nop
   nop
;****************************************************************
; Upon reset all ports are set to input and port G output is set
; low (0x0000).  Therefore it remains only to enable the pullups
; on port F.
;****************************************************************
   in  a0,IntGenCtrl
   or  a0,PFPULLUPS        ;enable port F pullups
   and a0,~TIM2IMR         ;turn off TIMER 2 interrupt
   and a0,~TIM2ENABLE         ;turn off TIMER 2 --- added 28.11.99
   out IntGenCtrl,a0
;****************************************************************
; Set the DAC to 10 bits, C3x style.  For C5x style set bit 3
; high.
;****************************************************************
   movb   a0,0x02             ;choose 10 bit DAC, C3x style
   orb a0,DACON         ;enable DAC
   out DACCTRL,a0          ;switch DAC on
;****************************************************************
; Initialization complete.  Now tidy up and branch to the main
; user code.
;****************************************************************
   zac a0           ;tidy up
   zac a0~

   jmp _main           ;jump to the main program
```

6.3 TI-TALKS Example Code

The TI-TALKS code contains the four vocoders (MELP, CELP, ADPCM, and LPC) and demonstrates how to use the interrupts to scan the keys and flash the LEDs. An LCD driver module is also included.

TI-TALKS should be used as a starting point for code development. Updates to the vocoders and other modules are sent out by Texas Instruments as necessary.

Please contact the TI speech applications group (email: **Speak2Me@list.ti.com**) for the latest version of the TI-TALKS example code.

Getting Started

Connect the MSP scan port (the small grey metal box) to the PC and to the speech development board. Ensure that the scan port and the development board are powered on (the red LED and the green LED on the scan port are both illuminated) before attempting to start the code development tool.

Click on Start, go to Programs – EMUC6xx and click on MSP50C6xx Code Development icon. To open a project click on *Project – New Project* and select the desired project file. e.g.,

C:\614\PROJECTS\TI–TALKS604\TI60OBJ.RPJ.

Note that this is an example for TI-TALKS code version 604. The file extension for the project file is RPJ.

Click on *Project – Build* to assemble and link the constituent files of the project. Then click *Debug – Eprom Programming* and select *Blank Check + Program* to burn the code onto a P614 device. Alternatively, press F3 then Enter.

Set the breakpoint at the `_main` label. To do this click on the blue magnifying glass icon at the top of the screen, then from the Symbol list choose `_main`. Click OK and the Program Window will display the label and the surrounding code. The line of code at `_main – MOV R7,STACK –` is highlighted in cyan. Set the breakpoint by moving the mouse to this line, holding the SHIFT key and clicking the right mouse button.

Click on *Init – Init All* to reset the P614. All the values in the RAM window should turn blue and should be zero (0000).

To run the program, click on the yellow lightning/black centipede (Run Internal) icon at the end of the tool bar. The program should halt at the `_main` label. All the values in the CPU window should be blue and zero apart from PC, STAT, DP, RZF and ZF.

To continue, click on the Run Internal icon again. A bugle call is synthesized in CELP and then the program loops round continuously.

Creating a New Project

The easiest way to create a new project is to copy the entire TI–TALKS604 directory into another directory and renaming the project file as desired. It is not necessary to change the paths of the files in the project – this will be done automatically by the code development tool. Note that TI-TALKS604 indicates version 604 of TI-TALKS code.

6.4 RAM Overlay

The RAM map for the MSP50C6xx family is quite complex. Here the method of overlaying the RAM is explained, together with examples of how to add variables for customer code.

6.4.1 RAM Usage

Information about the RAM overlay is contained in the following three include files (.IRX).

- MAIN_RAM.IRX
- RAM.IRX
- SPK_RAM.IRX

MAIN_RAM.IRX contains definitions for customer RAM. Variable and RAM for other modules (in the form of RAM.IRX files – see below) should be added here.

RAM.IRX contains definitions for the RAM used by the coders (MELP, CELP, LPC, and ADPCM). The only constants which should be changed by the user are STACK and RAMEND_DSP. The former defines the size of the stack, which is 20 words by default. The latter defines the amount of RAM consumed by the largest coder in use, and hence defines the location of the beginning of customer RAM. For example, if a program uses both the MELP and CELP coders, then RAMEND_DSP must be equal to RAMEND_MELP. If CELP and ADPCM are being used, the RAMEND_DSP must be set to RAMEND_CELP.

SPK_RAM.IRX contains definitions for the RAM used by the coders. Three of these variable, TEMP1, TEMP2, and TEMP3 may be used as general purpose temporary variables. SPK_RAM.IRX should never be edited or modified in anyway.

6.4.2 RAM Overlay

RAM is reserved for variables in the following way. The start address of the variable is equal to the address of the previous variable, plus the size of that variable. The size of VAR1 thus depends on the start address of the next variable. In the example below, dac_buffer starts 2 bytes (one word) after current_buffer. This means that current_buffer must be one word long. The variable after dac_buffer, save_dac_r0, starts 2 bytes (one word) after dac_buffer. Therefore, dac_buffer is one word long. Similarly, save_dac_stat starts 10 bytes (5 words) after save_dac_regs, therefore, save_dac_regs is a variable five words long.

```
dac_buffer    equ    current_buffer + 2 * 1                ;RESW     1
save_dac_r0   equ    dac_buffer + 2 * 1                    ;RESW     1
save_dac_regs equ    save_dac_r0 + 2 * 1                   ;RESW     5
save_dac_stat equ    save_dac_regs + 2 * 5                 ;RESW     1
```

The above method should be used to declare all customer variables. This is illustrated in the next section.

6.4.3 Adding Customer Variables

New variables should either be added directly to MAIN_RAM.IRX or should be included as a module RAM.IRX file. To add a variable new_var, size one word, would require adding the variable itself and modifying the RAMEND_CUSTOMER constant. The original MAIN_RAM.IRX file is shown below.

```
;************************************************************
; MAIN_RAM.IRX
;
; Start of memory for MAIN module is defined in
;            include"..\ram\ram.irx"
;************************************************************

; General purpose variables

ledpattern          equ    RAMSTART_CUSTOMER + 2 * 1
keypress            equ    ledpattern + 2 * 1
tabadr              equ    leypress + 2 * 1

; Time 1 interrupt variables

save_tim1_stat      equ    save_tim1_a0a + 2 * 1
save_tim2_a0        equ    save_tim2_stat + 2 * 1
save_tim2_a0a       equ    save_tim2_a0 + 2 * 1

; Time 2 interrupt variables
```

```
save_tim2_stat      equ    save_tim1_a0a + 2 * 1
save_tim2_a0        equ    save_tim2_stat + 2 * 1
save_tim2_a0a       equ    save_tim2_a0 + 2 * 1

;End of RAM

RAMEND_CUSTOMER     equ    save_tim2_a0a
RAMLENGTH_CUSTOMER  equ    RAMEND_CUSTOMER –
RAMSTART_CUSTOMER
```

After adding new_var the MAIN_RAM.IRX file would look like this:

```
;*************************************************************
; MAIN_RAM.IRX
;
; Start of memory for MAIN module is defined in
;             include"..\ram\ram.irx"
;*************************************************************

; General purpose variables

ledpattern          equ    RAMSTART_CUSTOMER + 2 * 1
keypress            equ    ledpattern + 2 * 1
tabadr              equ    leypress + 2 * 1

; Time 1 interrupt variables

save_tim1_stat      equ    save_tim1_a0a + 2 * 1
save_tim2_a0        equ    save_tim2_stat + 2 * 1
save_tim2_a0a       equ    save_tim2_a0 + 2 * 1

; Time 2 interrupt variables

save_tim2_stat      equ    save_tim1_a0a + 2 * 1
save_tim2_a0        equ    save_tim2_stat + 2 * 1
save_tim2_a0a       equ    save_tim2_a0 + 2 * 1

; End of RAM

RAMEND_CUSTOMER     equ    new_var
RAMLENGTH_CUSTOMER  equ    RAMEND_CUSTOMER –
RAMSTART_CUSTOMER
```

6.4.4 Common Problems

Since the location and size of a variable depends on a previously declared variable, it is possible to misspell a variable and end up with one or more variables starting at the wrong address. Therefore, it is worthwhile checking the MAIN.LST file and searching for RAMSTART_CUSTOMER, to ensure that all the customer variables are at the proper address.

Also, when modifying MAIN_RAM.IRX or any of the module RAM.IRX files, it is a good idea to *build* the project, rather than doing a *make*.

Chapter 7

Customer Information

Customer information regarding package configurations, development cycle, and ordering forms are included in this chapter.

Topic **Page**

7.1	Mechanical Information	7–2
7.2	Customer Information Fields in the ROM	7–11
7.3	Speech Development Cycle	7–12
7.4	Device Production Sequence	7–12
7.5	Ordering Information	7–14
7.6	New Product Release Forms	7–14

7.1 Mechanical Information

The MSP50C614, MSP50C605, and the MSP50C601 are normally sold in die form, but are also available in a 100-pin QFP package. The MSP50C604 is a available in die form and in a 64-pin QFP package. The MSP50P614 is available in a 120-pin, PGA-windowed ceramic package.

NOTE: Scan Port Bond Out

The scan port interface on the MSP50C6xx devices has five dedicated pins and one shared pin that need to be used by the MSP50Cxx code development tools. The SCANIN, SCANOUT, SCANCLK, SYNC, and TEST pins are dedicated to the scan port interface. The $\overline{\text{RESET}}$ pin is shared with the application. These pins may play an important role in debugging any system problems. For this reason, these pins must be bonded out on any MSP50C614 production board. Furthermore, it is recommended that these pins be connected to test points, so the development tool can be connected. Since the development tool requires V_{DD} and V_{SS}, test points connected to these signals are also needed.

The application circuits appearing in section 6.1 show the minimum recommended configuration for any MSP50C614 application board. For production purposes, the 1-kΩ resistor which appears at the $\overline{\text{RESET}}$ pin is optional. It is required for use with the scan port interface, but they may be shorted otherwise. The footprints for this resistor are strongly recommended.

7.1.1 Die Bond-Out Coordinates

Die bond-out coordinates are available upon request from Texas Instruments (email: **speak2me@list.ti.com**).

7.1.2 Package Information

The MSP50C614, MSP50C605, and the MSP50C601 are available in the 100-pin QFP package. See Figure 7–1 and Tables 7–1 thru 7–3. The MSP50C604 is a available the 64-pin QFP package. See Figure 7–2 and Table 7–4. For more detailed information, please refer to the device datasheets available on the TI speech web site (http://www.ti.com/sc/speech).

Table 7–1. Signal and Pad Descriptions for the MSP50C614

SIGNAL	PIN NUMBER	PAD NUMBER	I/O	DESCRIPTION
Input/Output Ports				
PA0 – PA7	66 – 59	75 – 68	I/O	Port A general-purpose I/O (1 Byte)
PB0 – PB7	76 – 69	85 – 78	I/O	Port B general-purpose I/O (1 Byte)
PC0 – PC7	90 – 83	8 – 1	I/O	Port C general-purpose I/O (1 Byte)
PD0 – PD7	100 – 93	18 – 11	I/O	Port D general-purpose I/O (1 Byte)
PE0 – PE7	51 – 44	63 – 56	I/O	Port E general-purpose I/O (1 Byte)
PF0 – PF7	16 – 9	31 – 24	I	Port F dedicated input (1 Byte)
PG0 – PG7	37 – 30	49 – 42	O	Port G dedicated output (1 Byte)
PG8 – PG15	25 – 18	39 – 32	O	Port G dedicated output (1 Byte)
Pins **PD4** and **PD5** may be dedicated to the comparator function, if the comparator enable bit is set. Refer to Section 3.3, *Comparator*, for details.				
Scan Port Control Signals				
SCANIN	42	54	I	Scan port data input
SCANOUT	38	50	O	Scan port data output
SCANCLK	41	53	I	Scan port clock
SYNC	40	52	I	Scan port synchronization
TEST	39	51	I	MSP50C6xx: test modes
The scan port pins must be bonded out on any MSP50C6xx production board. Consult the “Important Note regarding Scan Port Bond Out”.				
Reference Oscillator Signals				
OSCOUT	56	65	O	Resistor/crystal reference out
OSCIN	57	66	I	Resistor/crystal reference in
PLL	58	67	O	Phase-lock-loop filter
Digital-to-Analog Sound Outputs				
DACP	7	22	O	Digital-to-analog plus output (+)
DACM	5	20	O	Digital-to-analog minus output (–)
Initialization				
$\overline{\text{RESET}}$	43	55	I	Initialization
Power Signals				
V_{SS}	1†, 26, 52, 67, 91	9, 19†, 40, 64, 76		Ground
V_{DD}	6†, 8, 27, 68, 92	10, 21†, 23, 41, 77		Processor power (+)

† The V_{SS} and V_{DD} connections service the DAC circuitry. Their pins tend to sustain a higher current draw. A dedicated decoupling capacitor across these pins is therefore required.

Table 7–2. Signal and Pad Descriptions for the MSP50C605

SIGNAL	PIN NUMBER	PAD NUMBER	I/O	DESCRIPTION	
Input/Output Ports					
PC0 – PC7	89 – 82	8 – 1	I/O	Port C general-purpose I/O	(1 Byte)
PD0 – PD7	99 – 92	18 – 11	I/O	Port D general-purpose I/O	(1 Byte)
PE0 – PE7	46 – 39	48 – 41	I/O	Port E general-purpose I/O	(1 Byte)
PF0 – PF7	16 – 9	31 – 24	I	Port F dedicated input	(1 Byte)
Pins PD_4 and PD_5 may be dedicated to the comparator function, if the comparator enable bit is set. Refer to Section 3.3, *Comparator*, for details.					
Scan Port Control Signals					
SCANIN	37	39	I	Scan port data input	
SCANOUT	33	35	O	Scan port data output	
SCANCLK	36	38	I	Scan port clock	
SYNC	35	37	I	Scan port synchronization	
TEST	34	36	I	C605: test modes	
The scan port pins must be bonded out on any MSP50C605 production board. Consult the “Important Note regarding Scan Port Bond Out”.					
Reference Oscillator Signals					
OSCOUT	49	51	O	Resistor/crystal reference out	
OSCIN	48	50	I	Resistor/crystal reference in	
PLL	47	49	O	Phase-lock-loop filter	
Digital-to-Analog Sound Outputs					
DACP	7	22	O	Digital-to-analog plus output (+)	
DACM	5	20	O	Digital-to-analog minus output (–)	
Initialization					
$\overline{\text{RESET}}$	38	40	I	Initialization	
Power Signals					
V_{SS}	17, 50, 90, 100†	32, 52, 9, 19†		Ground	
V_{DD}	6†, 8, 31, 32, 91	21†, 23, 33, 34, 10		Processor power (+)	

† The V_{SS} and V_{DD} connections service the DAC circuitry. Their pins tend to sustain a higher current draw. A dedicated decoupling capacitor across these pins is therefore required.

Table 7–3. Signal and Pad Descriptions for the MSP50C601

SIGNAL	PIN NUMBER	PAD NUMBER	I/O	DESCRIPTION	
Input/Output Ports					
PC0 – PC7	89 – 82	8 – 1	I/O	Port C general-purpose I/O	(1 Byte)
PD0 – PD7	99 – 92	18 – 11	I/O	Port D general-purpose I/O	(1 Byte)
PE0 – PE7	46 – 39	48 – 41	I/O	Port E general-purpose I/O	(1 Byte)
PF0 – PF7	16 – 9	31 – 24	I	Port F dedicated input	(1 Byte)
Pins PD_4 and PD_5 may be dedicated to the comparator function, if the comparator enable bit is set. Refer to Section 3.3, *Comparator*, for details.					
Scan Port Control Signals					
SCANIN	37	39	I	Scan port data input	
SCANOUT	33	35	O	Scan port data output	
SCANCLK	36	38	I	Scan port clock	
SYNC	35	37	I	Scan port synchronization	
TEST	34	36	I	C605: test modes	
The scan port pins must be bonded out on any MSP50C601 production board. Consult the “Important Note regarding Scan Port Bond Out”.					
Reference Oscillator Signals					
OSCOUT	49	51	O	Resistor/crystal reference out	
OSCIN	48	50	I	Resistor/crystal reference in	
PLL	47	49	O	Phase-lock-loop filter	
Digital-to-Analog Sound Outputs					
DACP	7	22	O	Digital-to-analog plus output (+)	
DACM	5	20	O	Digital-to-analog minus output (–)	
Initialization					
$\overline{\text{RESET}}$	38	40	I	Initialization	
Power Signals					
V_{SS}	17, 50, 90, 100†	32, 52, 9, 19†		Ground	
V_{DD}	6†, 8, 31, 32, 91	21†, 23, 33, 34, 10		Processor power (+)	

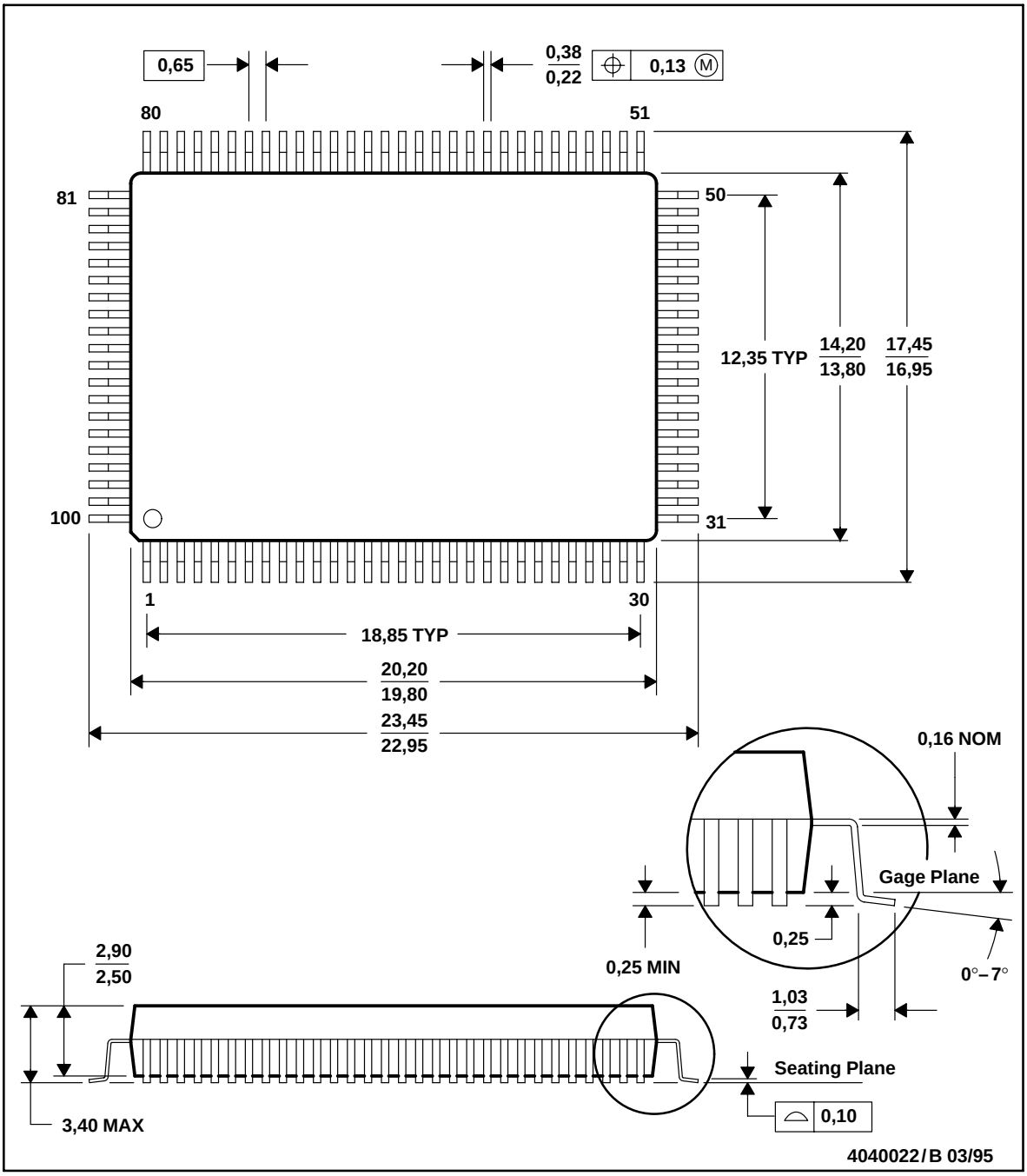
† The V_{SS} and V_{DD} connections service the DAC circuitry. Their pins tend to sustain a higher current draw. A dedicated decoupling capacitor across these pins is therefore required.

Table 7–4. Signal and Pad Descriptions for the MSP50C604

SIGNAL	PIN NUMBER	PAD NUMBER	I/O	DESCRIPTION	
Input/Output Ports					
PC0 – PC7	89 – 82	8 – 1	I/O	Port C general-purpose I/O	(1 Byte)
PD0 – PD7	99 – 92	18 – 11	I/O	Port D general-purpose I/O	(1 Byte)
PE0 – PE7	46 – 39	48 – 41	I/O	Port E general-purpose I/O	(1 Byte)
PF0 – PF7	16 – 9	31 – 24	I	Port F dedicated input	(1 Byte)
Pins PD_4 and PD_5 may be dedicated to the comparator function, if the comparator enable bit is set. Refer to Section 3.3, *Comparator*, for details.					
Scan Port Control Signals					
SCANIN	37	39	I	Scan port data input	
SCANOUT	33	35	O	Scan port data output	
SCANCLK	36	38	I	Scan port clock	
SYNC	35	37	I	Scan port synchronization	
TEST	34	36	I	C605: test modes	
The scan port pins must be bonded out on any MSP50C604 production board. Consult the “Important Note regarding Scan Port Bond Out”.					
Reference Oscillator Signals					
OSCOUT	49	51	O	Resistor/crystal reference out	
OSCIN	48	50	I	Resistor/crystal reference in	
PLL	47	49	O	Phase-lock-loop filter	
Digital-to-Analog Sound Outputs					
DACP	7	22	O	Digital-to-analog plus output (+)	
DACM	5	20	O	Digital-to-analog minus output (–)	
Initialization					
$\overline{\text{RESET}}$	38	40	I	Initialization	
Power Signals					
V_{SS}	17, 50, 90, 100†	32, 52, 9, 19†		Ground	
V_{DD}	6†, 8, 31, 32, 91	21†, 23, 33, 34, 10		Processor power (+)	

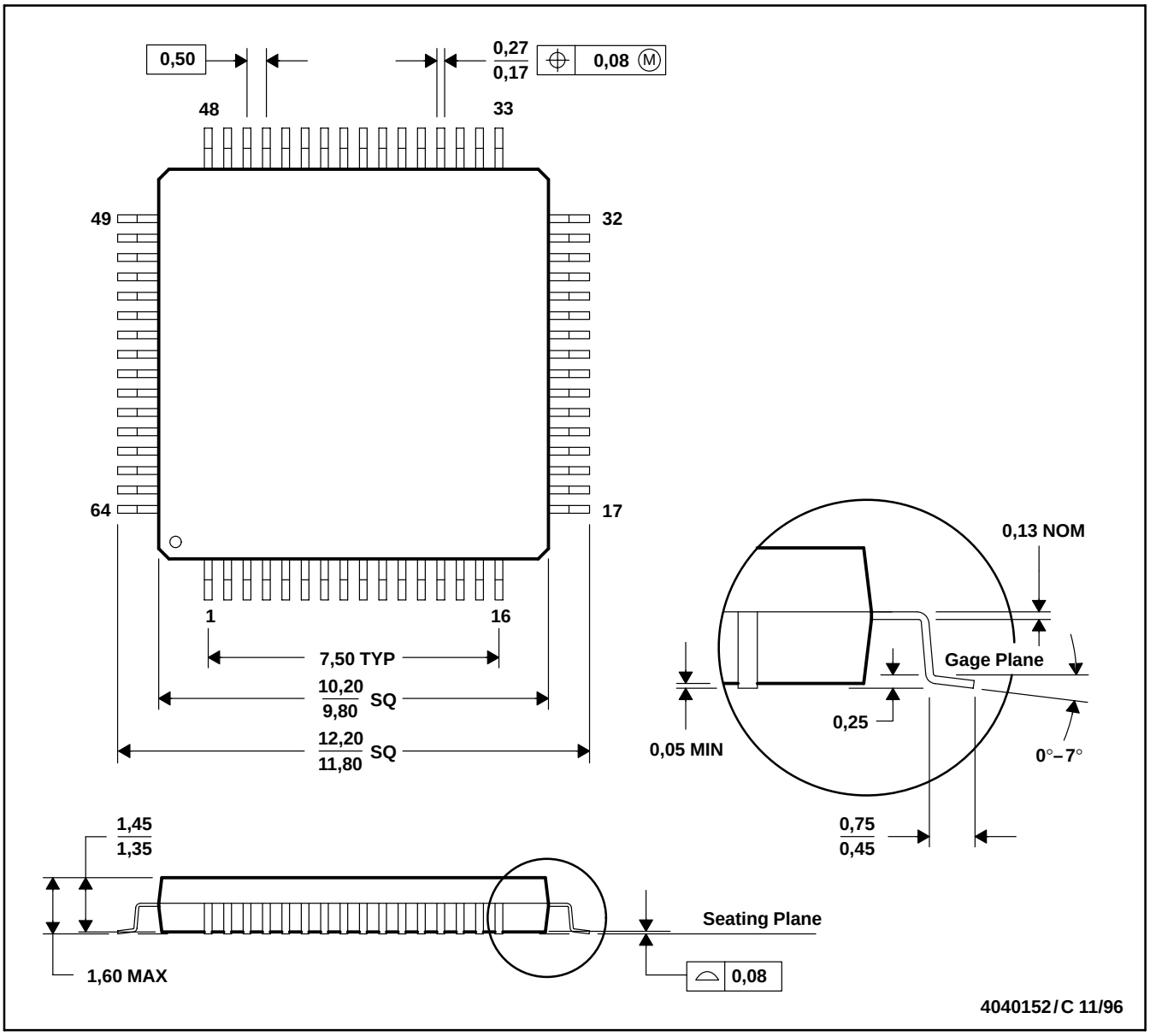
† The V_{SS} and V_{DD} connections service the DAC circuitry. Their pins tend to sustain a higher current draw. A dedicated decoupling capacitor across these pins is therefore required.

Figure 7–1. 100-Pin QFP Mechanical Information



NOTES: A. All linear dimensions are in millimeters.
B. This drawing is subject to change without notice.
C. Falls within JEDEC MS-022

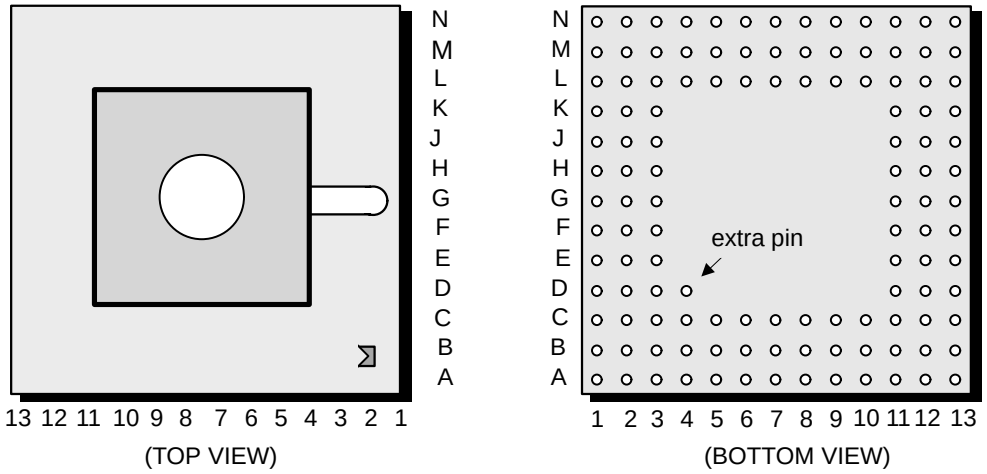
Figure 7–2. 64-Pin QFP Mechanical Information



NOTES: A. All linear dimensions are in millimeters.
B. This drawing is subject to change without notice.
C. Falls within JEDEC MS-026
D. May also be thermally enhanced plastic with leads connected to the die pads.

The MSP50C614 is available in a windowed-ceramic, 120-pin, grid array (PGA) packaged for use in software development and prototyping. This PGA package is shown in Figure 7–3.

Figure 7–3. 120-Pin, Grid Array Package for the Development Device, MSP50P614



Note:

The PGA package is only available in limited quantities for development purposes.

The pin assignments for the 120-pin PGA are outlined in Figure 7–4.

Figure 7–4. Bottom View of 120-Pin PGA Package of the MSP50P614

	1	2	3	4	5	6	7	8	9	10	11	12	13
N	nc	nc	†V_{DD}	PF_7	PF_5	PF_2	V_{PP}	PG_{15}	PG_{12}	PG_{10}	V_{SS}	V_{DD}	nc
M	nc	nc	DACM	DAC_P	PF_6	PF_3	PF_1	PG_{14}	PG_{11}	PG_8	nc	nc	PG_7
L	PD_0	nc	nc	†V_{SS}	V_{DD}	PF_4	PF_0	PG_{13}	PG_9	nc	nc	PG_5	PG_4
K	PD_3	PD_1	nc								PG_6	PG_3	PG_1
J	PD_5	PD_4	PD_2								PG_2	PG_0	scanout
H	V_{DD}	PD_7	PD_6								pgmpuls	SYNC	scanclk
G	V_{SS}	PC_1	PC_0			**(bottom view)**					$\overline{\text{RESET}}$	scanin	PE_7
F	PC_2	PC_3	PC_4								PE_4	PE_5	PE_6
E	PC_5	PC_6	nc								PE_0	PE_2	PE_3
D	PC_7	nc	nc	extra							nc	V_{SS}	PE_1
C	nc	nc	nc	nc	PB_1	PB_5	V_{SS}	PA_3	PA_7	nc	nc	nc	nc
B	nc	nc	nc	PB_0	PB_3	PB_6	PA_0	PA_2	PA_5	PLL	OSC_{OUT}	nc	nc
A	nc	nc	nc	PB_2	PB_4	PB_7	V_{DD}	PA_1	PA_4	PA_6	OSC_{IN}	nc	nc

† It is important to provide a separate decoupling capacitor for the $\mathbf{V_{DD}}$, $\mathbf{V_{SS}}$ pair which services the DAC. These pins are PGA numbers *N3* and *L4*, respectively. The relatively high current demands of the digital-to-analog circuitry make this a requirement. Refer to section 6.1 for details.

7.2 Customer Information Fields in the ROM

Customer code information is inserted in the ROM by Texas Instruments. This information appears as seven distinct fields within the ROM test-area. The ROM test-area extends from address 0x0000 to 0x07FF. The code-release information is stored in locations 0x0006 through 0x000C. Assuming these addresses are not specifically read-protected by the ROM security, they are read-accessible to the programmer. The fields appear as follows:

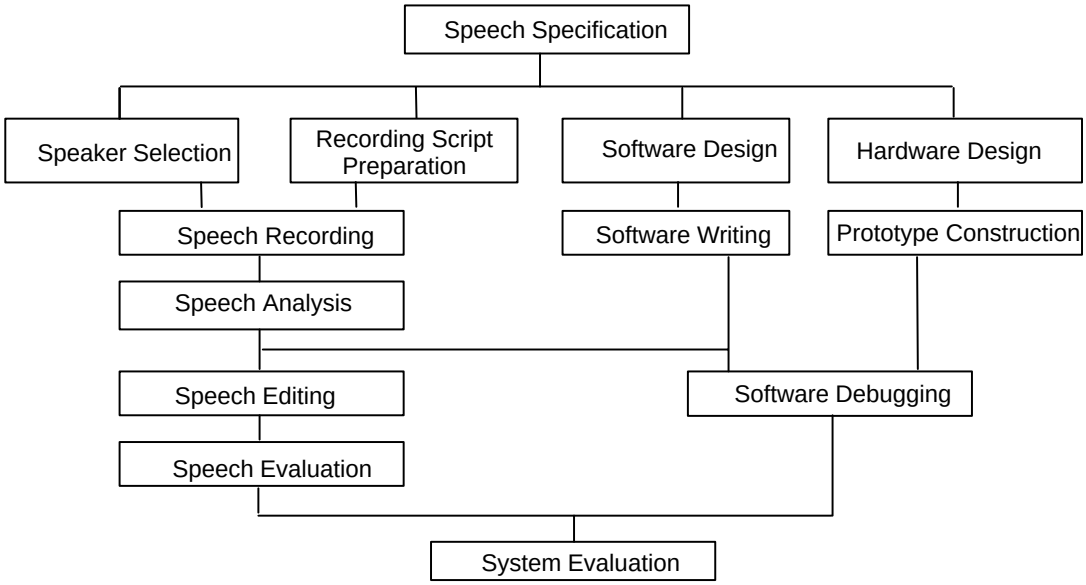
MSP50C614 EPROM Test-Area
Customer Information Fields (16-bit wide, the 17th bit is ignored)

Address	Field Description	Example Value
0x0006	Device number	0x0614 (for MSP50C614)
0x0007	Mask number (assigned by TI)	0x0005
0x0008	Reserved	
0x0009	Customer code version number	0x0001
0x000A	Customer code revision number	0x0005 (e.g., version 1.5)
0x000B	Year mask generated	0x1999
0x000C	Data mask generated (mm/dd)	0x0816 (e.g., 8/16/1999)

7.3 Speech Development Cycle

A sample speech development cycle is shown in Figure 7–5. Some of the components, such as speech recording, speech analysis, speech editing, and speech evaluation, require different hardware and software. TI provides a speech development tool, called the SDS6000, which allows the user to perform speech analysis using various algorithms, speech editing for certain algorithms, and to evaluate synthesis results through playback of encoded speech. Design of the software and hardware, development of software, and prototype construction are all customer-dependent aspects of the speech development cycle.

Figure 7–5. Speech Development Cycle



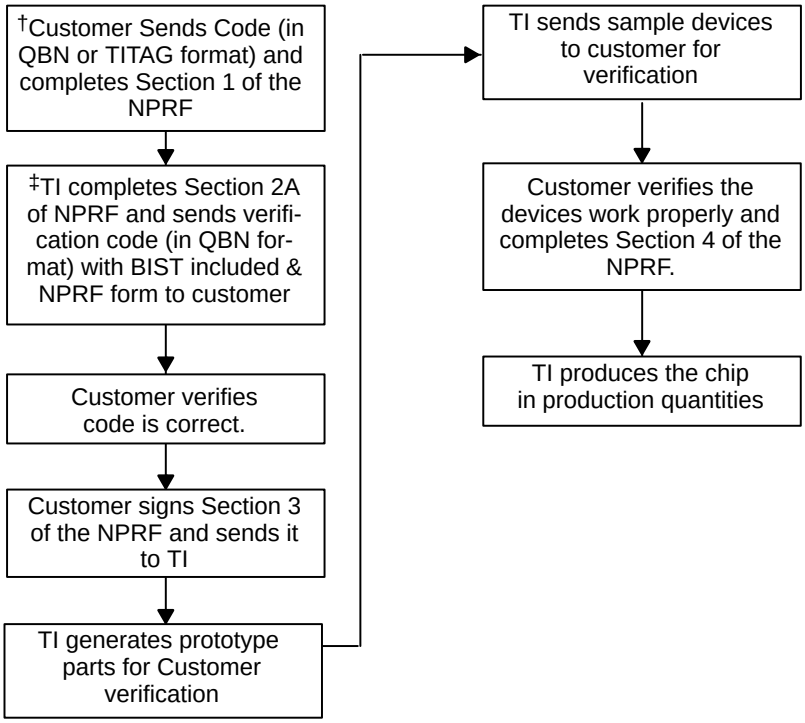
7.4 Device Production Sequence

For the speech development group at TI to accept a custom device program, the customer must submit a new product release form (NPRF). This form describes the custom features of the device (e.g., customer information, prototype and production qualities, symbolization, etc.). Section 1 is completed by the customer and Section 2B is completed by the customer for package sales. Section 2A is completed by TI personnel. A copy of the NPRF can be found in section 7.6. Copies can be downloaded at **www.ti.com/sc/speech**.

TI generates the prototype photomask, then processes, manufactures, and tests prototype devices for shipment to the customer. The number of prototypes is 25, for package sales and 200 for die sales, plus additional units if requested.

All prototype devices are shipped with the following disclaimer: *It is understood that, for expediency purposes, the initial 25 prototype devices (and any additional prototype devices purchased) were assembled on a prototype (i.e., not production-qualified) manufacturing line, whose reliability has not been characterized. Therefore, the anticipated inherent reliability of these devices cannot be expressly defined.*

The customer verifies the operation and quality of the prototypes and responds with either written customer prototype approval or disapproval. A nonrecurring mask charge that includes the 25 prototype devices is incurred by the customer. A minimum purchase is required during the first year of production.

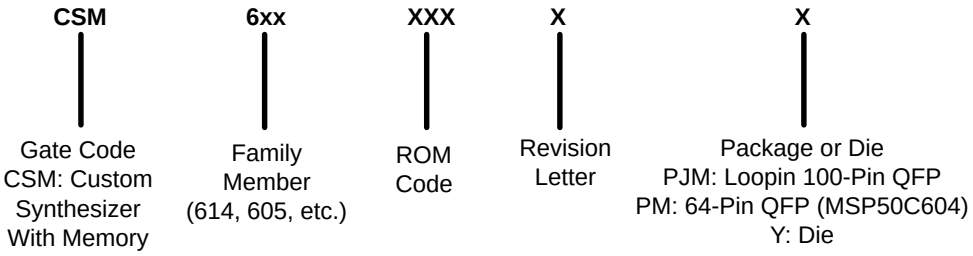


† For MSP50C601 and MSP50C605 devices, the customer needs to send the code in QBN format and speech data in BIN format.
‡ For MSP50C601 and MSP50C605 devices, Texas Instruments will send the verification code in QBN format and the verification speech data in BIN format.

Texas Instruments recommends that prototype devices not be used in production systems. The expected end-use failure rate of these devices is undefined; however, it is predicted to be greater than that of the standard qualified production.

7.5 Ordering Information

Because the MSP50C6xx are custom devices, they receive a distinct identification, as follows:



7.6 New Product Release Forms (NPRF)

The new product release form is used to track and document all the steps involved in implementing a new speech code onto one of the parent speech devices.

Section 1 of the NPRF is completed by the customer (and section 2B if for package sales) and sent to TI with the code.

Please refer to Section 7.4 of the manual for more information on device production sequence.

The following are the NPRFs for the MSP50C614, MSP50C604, MSP50C605, and the MSP50C601.

NEW PRODUCT RELEASE FORM FOR MSP50C614

SECTION 1. OPTION SELECTION

This section is to be completed by the customer and sent to TI along with the microprocessor code and speech data.

Company:_____ Division:_____

Project Name:_____ Purchase Order #:_____

Management Contact:_____ Phone:(___)_____

Technical Contact:_____ Phone:(___)_____

Customer Part Number:_____

Customer Code Version and Revision: __.__ (of format vv.rr)

(vv = version, rr = revision; numeric values only)

Package Type (check one):

___ PJM (100 pin QFP)

___ Die

****

SECTION 2A. ASSIGNMENT OF TI PRODUCTION PART NUMBER

This section is to be completed by TI.

TI Part Number: _____ (CSM614xxxY or CSM614xxxPJM)

SECTION 2B. PACKAGE UNIT SYMBOLIZATION

This section is to be completed by the customer. The first line of the symbolization is fixed. Except EIA#/Logo. The second and third lines are to be filled in by the customer.

Top Side Symbolization (100pin 'PJM')

```
+----------------------+   LLLL: LOT TRACE CODE
|     ??? YMLLLLT      |     YM: DATE CODE
|    <optional 13 char> |      T: ASSY SITE
|    <optional 12 char> |    ???: TI EIA NO. or
+----------------------+           TI LOGO
```

For '100 PJM' package the customer may choose between

TI EIA No. 980 or the TI LOGO on the first line.

2nd Line is typically the TI Part Number.

****

SECTION 3. AUTHORIZATION TO GENERATE MASKS, PROTOTYPES, AND RISK UNITS

This section is to be completed by the customer and sent to TI after the following criteria have been met:

1) The customer has verified that the TI computer generated data matches the original data.

2) The customer approves of the symbolization format in Section 2B. (Applies to packaged devices only).

I hereby certify that the TI generated verification data has been checked and found to be correct, and I authorize TI to generate masks, prototypes, and risk units in accordance with purchase order in section 1 above. In addition, in the instance that this is a packaged device, I also authorize TI to use the symbolization format illustrated in section 2B on all devices.

By:_____ Title:_____

Date:_____

(FAX this form to 214–480–7301. Attn: Code Release Team)

****

SECTION 4. APPROVAL OF PROTOTYPES AND AUTHORIZATION TO START PRODUCTION

This section is to be completed by the customer after prototype devices have been received and tested.

I hereby certify that the prototype devices have been received and tested and found to be acceptable, and I authorize TI to start normal production in accordance with purchase order #_____.

By:_____ Title:_____

Date:_____

****

Return to: Texas Instruments, Inc.
Attn: Code Release Team
P.O. Box 660199, M/S 8718
Dallas, TX 75266–0199

OR Fax to: (214)480–7301
Attn: Code Release Team

Have Questions?:

CALL: Code Release Team
(214)480–4444

OR E–MAIL: code–rel@msp.sc.ti.com

NEW PRODUCT RELEASE FORM FOR MSP50C604

SECTION 1. OPTION SELECTION

This section is to be completed by the customer and sent to TI along with the microprocessor code and speech data.

Company:_____ Division:_____

Project Name:_____ Purchase Order #:_____

Management Contact:_____ Phone:(___)_____

Technical Contact:_____ Phone:(___)_____

Customer Part Number:_____

Customer Code Version and Revision: __.__ (of format vv.rr)

(vv = version, rr = revision; numeric values only)

Package Type (check one):

___ PM (64 Pin)

___ die

Customer Code Version and Revision: __.__ (of format vv.rr)

(vv = version, rr = revision; numeric values only)

****

SECTION 2A. ASSIGNMENT OF TI PRODUCTION PART NUMBER

This section is to be completed by TI.

TI Part Number: _____ (CSM604xxxY or CSM604xxxPM)

SECTION 2B. PACKAGE UNIT SYMBOLIZATION

This section is to be completed by the customer. The first line of the symbolization is fixed. Except EIA#/Logo. The second and third lines are to be filled in by the customer.

Top Side Symbolization (64pin 'PM')

```
+----------------------+   LLLL: LOT TRACE CODE
|     ??? YMLLLLT      |     YM: DATE CODE
|    <optional 10 char> |      T: ASSY SITE
|    <optional 10 char> |    ???: TI EIA NO. or
+----------------------+           TI LOGO
```

For '64 PM' package the customer may choose between TI EIA No. 980 or the TI LOGO on the first line.

****

SECTION 3. AUTHORIZATION TO GENERATE MASKS, PROTOTYPES, AND RISK UNITS

This section is to be completed by the customer and sent to TI after the following criteria have been met:

1) The customer has verified that the TI computer generated

data matches the original data.

2) The customer approves of the symbolization format in Section 2B. (Applies to packaged devices only).

I hereby certify that the TI generated verification data has been checked and found to be correct, and I authorize TI to generate masks, prototypes, and risk units in accordance with purchase order in section 1 above. In addition, in the instance that this is a packaged device, I also authorize TI to use the symbolization format illustrated in section 2B on all devices.

By:_____ Title:_____

Date:_____

(FAX this form to 214–480–7301. Attn: Code Release Team)

****

SECTION 4. APPROVAL OF PROTOTYPES AND AUTHORIZATION TO START PRODUCTION

This section is to be completed by the customer after prototype devices have been received and tested.

I hereby certify that the prototype devices have been received and tested and found to be acceptable, and I authorize TI to start normal production in accordance with purchase order #_____.

By:_____ Title:_____

Date:_____

****

Return to: Texas Instruments, Inc.
Attn: Code Release Team
P.O. Box 660199, M/S 8718
Dallas, TX 75266–0199

OR Fax to: (214)480–7301
Attn: Code Release Team

Have Questions?:

CALL: Code Release Team
(214)480–4444

OR E–MAIL: code–rel@msp.sc.ti.com

NEW PRODUCT RELEASE FORM FOR MSP50C605

SECTION 1. OPTION SELECTION

This section is to be completed by the customer and sent to TI along with the microprocessor code and speech data.

Company:_____ Division:_____

Project Name:_____ Purchase Order #:_____

Management Contact:_____ Phone:(___)_____

Technical Contact:_____ Phone:(___)_____

Customer Part Number:_____

Customer Code Version and Revision: __.__ (of format vv.rr)

(vv = version, rr = revision; numeric values only)

Package Type (check one):

___ PJM (100 pin QFP)

___ Die

****

SECTION 2A. ASSIGNMENT OF TI PRODUCTION PART NUMBER

This section is to be completed by TI.

TI Part Number: _____ (CSM605xxxY or CSM605xxxPJM)

SECTION 2B. PACKAGE UNIT SYMBOLIZATION

This section is to be completed by the customer. The first line of the symbolization is fixed. Except EIA#/Logo. The second and third lines are to be filled in by the customer.

Top Side Symbolization (100pin 'PJM')

```
+----------------------+   LLLL: LOT TRACE CODE
|     ??? YMLLLLT      |     YM: DATE CODE
|    <optional 13 char> |      T: ASSY SITE
|    <optional 12 char> |    ???: TI EIA NO. or
+----------------------+           TI LOGO
```

For '100 PJM' package the customer may choose between

TI EIA No. 980 or the TI LOGO on the first line.

2nd Line is typically the TI Part Number.

****

SECTION 3. AUTHORIZATION TO GENERATE MASKS, PROTOTYPES, AND RISK UNITS

This section is to be completed by the customer and sent to TI after the following criteria have been met:

1) The customer has verified that the TI computer generated data matches the original data.

2) The customer approves of the symbolization format in Section 2B. (Applies to packaged devices only).

I hereby certify that the TI generated verification data has been checked and found to be correct, and I authorize TI to generate masks, prototypes, and risk units in accordance with purchase order in section 1 above. In addition, in the instance that this is a packaged device, I also authorize TI to use the symbolization format illustrated in section 2B on all devices.

By:_____ Title:_____

Date:_____

(FAX this form to 214–480–7301. Attn: Code Release Team)

****

SECTION 4. APPROVAL OF PROTOTYPES AND AUTHORIZATION TO START PRODUCTION

This section is to be completed by the customer after prototype devices have been received and tested.

I hereby certify that the prototype devices have been received and tested and found to be acceptable, and I authorize TI to start normal production in accordance with purchase order #_____.

By:_____ Title:_____

Date:_____

****

Return to: Texas Instruments, Inc.
Attn: Code Release Team
P.O. Box 660199, M/S 8718
Dallas, TX 75266–0199

OR Fax to: (214)480–7301
Attn: Code Release Team

Have Questions?:

CALL: Code Release Team
(214)480–4444

OR E–MAIL: code–rel@msp.sc.ti.com

NEW PRODUCT RELEASE FORM FOR MSP50C601

SECTION 1. OPTION SELECTION

This section is to be completed by the customer and sent to TI along with the microprocessor code and speech data.

Company:_____ Division:_____

Project Name:_____ Purchase Order #:_____

Management Contact:_____ Phone:(___)_____

Technical Contact:_____ Phone:(___)_____

Customer Part Number:_____

Customer Code Version and Revision: __.__ (of format vv.rr)

(vv = version, rr = revision; numeric values only)

Package Type (check one):

___ PJM (100 pin QFP)

___ Die

****

SECTION 2A. ASSIGNMENT OF TI PRODUCTION PART NUMBER

This section is to be completed by TI.

TI Part Number: _____ (CSM601xxxY or CSM601xxxPJM)

SECTION 2B. PACKAGE UNIT SYMBOLIZATION

This section is to be completed by the customer. The first line of the symbolization is fixed. Except EIA#/Logo. The second and third lines are to be filled in by the customer.

Top Side Symbolization (100pin 'PJM')

```
+----------------------+   LLLL: LOT TRACE CODE
|     ??? YMLLLLT      |     YM: DATE CODE
|    <optional 13 char> |      T: ASSY SITE
|    <optional 12 char> |    ???: TI EIA NO. or
+----------------------+           TI LOGO
```

For '100 PJM' package the customer may choose between

TI EIA No. 980 or the TI LOGO on the first line.

2nd Line is typically the TI Part Number.

****

SECTION 3. AUTHORIZATION TO GENERATE MASKS, PROTOTYPES, AND RISK UNITS

This section is to be completed by the customer and sent to TI after the following criteria have been met:

1) The customer has verified that the TI computer generated data matches the original data.

2) The customer approves of the symbolization format in Section 2B. (Applies to packaged devices only).

I hereby certify that the TI generated verification data has been checked and found to be correct, and I authorize TI to generate masks, prototypes, and risk units in accordance with purchase order in section 1 above. In addition, in the instance that this is a packaged device, I also authorize TI to use the symbolization format illustrated in section 2B on all devices.

By:_____ Title:_____

Date:_____

(FAX this form to 214–480–7301. Attn: Code Release Team)

****

SECTION 4.APPROVAL OF PROTOTYPES AND AUTHORIZATION TO START PRODUCTION

This section is to be completed by the customer after prototype devices have been received and tested.

I hereby certify that the prototype devices have been received and tested and found to be acceptable, and I authorize TI to start normal production in accordance with purchase order #_____.

By:_____ Title:_____

Date:_____

****

Return to: Texas Instruments, Inc.
Attn: Code Release Team
P.O. Box 660199, M/S 8718
Dallas, TX 75266–0199

OR Fax to: (214)480–7301
Attn: Code Release Team

Have Questions?:

CALL: Code Release Team
(214)480–4444

OR E–MAIL: code–rel@msp.sc.ti.com

Appendix A

Additional Information

This appendix contains additional information for the MSP50C6xx mixed-signal processor.

Topic **Page**

A.1 Additional Information A–2

A.1 Additional Information

For current information regarding the MSP50C6xx devices (data sheets, development tools, etc.), visit the TI Speech Web site:

http://www.ti.com/sc/speech